



PATENT
Attorney's Matter No. P0925-RE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Reissue application of:
Geoffrey B. Rhoads

Examining Group: 2621

Application No.: 10/766,750

Filed: January 27, 2004

For: METHODS FOR SURVEYING
DISSEMINATION OF PROPRIETARY
EMPIRICAL DATA

Examiner: J. Couso

Date: November 16, 2004

CERTIFICATE OF MAILING

I hereby certify that this paper and the documents referred to as being attached or enclosed herewith are being deposited with the United States Postal Service on November 16, 2004, as First Class Mail in an envelope addressed to: MAIL STOP AMENDMENT, COMMISSIONER FOR PATENTS, P.O. BOX 1450, ALEXANDRIA, VA 22313-1450.

Joel R. Meyer
Attorney for Applicant

DECLARATION

1. The specification is amended by a Preliminary Amendment to include a source code listing originally filed in U.S. Application 08/649,419 as Appendix B.
2. Appendix B was originally filed in U.S. Application 08/649,419, on May 16, 1996.
3. The present computer program listing appendix (i.e., file Appendix B.txt) consists of the same material as the original Appendix B.
4. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Date: November 16, 2004

Customer Number 23735

Telephone: 503-469-4800
FAX: 503-469-4777

Respectfully submitted,

DIGIMARC CORPORATION

By

Joel R. Meyer
Registration No. 37,677

ALIGN.CPP

7

```

scale_increment*pow(1.0/(double)START_RADIUS, 1.0/(double)lp_sampling);
for(i=0;i<lp_sampling;i++){
    radius[i] = (START_RADIUS*(double)dim2) * pow(scale_increment, (double)i);
}

pout = out;
for(theta=0.0;j=0;j<lp_sampling; j++,theta += (PI/lp_sampling)){
    dx = cos(theta);
    dy = sin(theta);
    pradius = radius;
    pout = fout[j];
    for(i=0;i<lp_sampling;i++){
        x = (double)dim2 + pradius * dx;
        y = (pradius++) * dy;
        yy = (int)y;
        fracy = x * (double)xx;
        fracy = y * (double)yy;
        pin = sin[yy*dim + xx];
        *pout += (float) ((1.0-fracy)*(1.0-fracy)* (double)*(pin++));
        *pout += (float) ( fracy*(1.0-fracy)* (double)*pin );
        pin += (dim-1);
        *pout += (float) ( (1.0-fracy)*fracy* (double)*(pin++));
        *pout += (float) ( fracy*fracy * (double)*pin );
        *pout += lp_sampling;
    }
}

/* now filter it along the scale axis */
/* this generally increases the peak to noise ratio in finding the proper scale rotation */
for(i=0;i<lp_sampling;i++){
    pout = ftemp;
    for(j=0;j<lp_sampling;j++){
        *pout = (float)0.0;
        for(k=- (LOG_MOV_AVG/2); k<= (LOG_MOV_AVG/2); k++){
            if(j-k)>0;
            else if(j)>= lp_sampling) j=lp_sampling-1;
            *pout += out[i+j]*lp_sampling;
        }
        *pout += (float)LOG_MOV_AVG;
    }
    pin = ftemp;
    pout = fcout[i];
    for(j=0;j<lp_sampling;j++){
        *pout = (float)0.0;
        for(k=- (LOG_SMOOTH/2); k<= (LOG_SMOOTH/2); k++){
            if(j-k)>0;
            else if(j)>= lp_sampling) j=lp_sampling-1;
            *pout += out[i+j]*lp_sampling;
        }
        *pout += (float)LOG_SMOOTH;
    }
    memcpy(fcout[i], ftemp, lp_sampling*sizeof(float));
}
return(i);
}

float get_median_float(float *array, int xdim, int ydim, int high_x, int high_y,
float *x_offset, float *y_offset){
    int j, ftemp, k, ktemp;

    ymedian[0] = ymedian[1] = ymedian[2] = (float)0.0;
    xmedian[0] = xmedian[1] = xmedian[2] = (float)0.0;
    py = ymedian;
    for(j=-1;j<2;j++){
        ftemp = high_y+1;
        if(jtemp < 0) jtemp = ydim-1;
        else if(jtemp == ydim) jtemp = 0;
        px = xmedian;
        for(k=-1;k<2;k++){
            ktemp = high_x+k;

```

```

        if(ktemp < 0) ktemp = xdim-1;
        else if(ktemp == xdim) ktemp = 0;
        *py += array[jtemp*xdim+ktemp];
        *px++ += array[jtemp*xdim+ktemp];
    }
    py++;
}
/* now find median values */
ratio = get_median_float(ymedian);
*y_offset = (float)high_y + ratio;
ratio = get_median_float(xmedian);
*x_offset = (float)high_x + ratio;
value = (xmedian[0]*xmedian[1]+xmedian[2])/(float)9.0;
return(value);
}

/* this is the fft window profile for mitigating edge effects; change to other windows if
their better */
/* or.... maybe certain windows are better for certain tasks, e.g., log polar vs. straight
correlation */
int load_windowing_function(int dim, float *window){
    int i;
    double step, x, y;

    step = 2.0*PI / (double)(dim+1);
    for(i=0; i<dim; i++, x+=step){
        y = (1.0 - cos(x))/2.0;
        window[i] = (float)sqrt(y);
    }
    return(i);
}

int window_id_vector(
float *array,
int data_length,
int full_length
){
    int i;
    float *pararray, *pwindow;

    float *window_function = new float[data_length];
    load_windowing_function(data_length, window_function);
    pararray = array;
    pwindow = window_function;
    for(i=0; i<data_length; i++){
        *pararray++ = *pwindow++;
    }
    if(full_length != data_length){
        for(i=0; i<(full_length - data_length); i++){
            *pararray++ = (float)0.0;
        }
        delete[] window_function;
        return(i);
    }
}

/* this module specifically designed for the rough thumbnail registration
in an earlier version of this routine. It performed bi-linear interpolation
on the pixels, but now think this is overkill because of the later refinement
anyway, who knows */
int rotate_scale_translate_image(
float *out,
int outdim,
float *in,
int indim,
int inxdim,
int inydim,
int orig_xdim,
int orig_ydim,
int downsample,
float rotation,
float scale
){
    int i, j, xx, yy;
    float a_const, b_const, x, y, dx, dy, *pout;
    float middle_in_x, middle_in_y, middle_out;

    /* make sure to place the center of the original array at the center of
the output array; this helps later translation bookkeeping */
    middle_in_x = (float)(orig_xdim - downsample)/(float)downsample/(float)2.0;
    middle_in_y = (float)(orig_ydim - downsample)/(float)downsample/(float)2.0;
    middle_out = (float)(outdim-1)/(float)2.0;
    rotation = -rotation; // who can keep track of CW and CCW anyway???
    a_const = (float)cos((double)rotation*PI/180.0)*scale;
    b_const = (float)sin((double)rotation*PI/180.0)*scale;
    dx = a_const;
    dy = b_const;
    pout = out;
    for(i=0; i<outdim; i++){
        x = middle_in_x - a_const*middle_out + b_const*(middle_out - (float)i) + (float)0.5;
        y = middle_in_y - b_const*middle_out - a_const*(middle_out - (float)i) + (float)0.5;

```

```

for(j=0;j<outdim;j++){
  if(x<(float)0.5 || x>(float)(inxdim-1)*(float)0.5){y<(float)0.5 ||
    else {y>(float)(inydim-1)*(float)0.5}*(pout++)=(float)0.0;
  }
  xx = (int)x;
  yy = (int)y;
  *(pout++) = in[yy*outdim+xx];
  x+=dx;
  y+=dy;
}
}

return(1);
}

int gmf(
  float *real1,
  float *real2,
  int dim,
  int bits,
  int *number_candidates,
  float *x_offset,
  float *y_offset,
  float *value,
  int type
){
  int dim2 = dim/2,i,j,k,l,ok,jtemp,ktemp;
  int x_off[MAX_CANDIDATES],y_off[MAX_CANDIDATES];
  float mag1,mag2,dot,cross,highest,ratio,ymedian[3],xmedian[3],*py,*px;
  float *preal1,*preal2,*pimaginary1,*pimaginary2;
  float tmp,dott;

  /* calculate phase differences and reload them into real1 and imaginary1 */
  preal1=real1,pimaginary1=real1[dim];
  preal2=real2,pimaginary2=real2[dim];
  for(i=0;i<(1+dim/2);i++){
    for(j=0;j<dim;j++){
      mag1 = (float)sqrt( (double) (*preal1 + *pimaginary1 * *pimaginary1) );
      mag2 = (float)sqrt( (double) (*preal2 + *pimaginary2 * *pimaginary2) );
      if(mag1 == (float)0.0)mag1=(float)SMALL;
      if(mag2 == (float)0.0)mag2=(float)SMALL;
      dott = (*preal1 * *preal2 + *pimaginary1 * *pimaginary2)/mag1/mag2;
      if(dott<(float)1.0 - dot*dot;
      dott = (float)sqrt( (double)dott );
      cross = *preal1 * *pimaginary2 - *pimaginary1 * *preal2;
      if(cross < (float)0.0)cross = -(float)1.0;
      else cross = (float)1.0;
      ftmp = mag2;
      dott*=ftmp;dott*=ftmp;
      *(preal1++) = dott;
      *(pimaginary1++) = cross*dott;
    }
    preal1+=dim;
    pimaginary1+=dim;
    preal2+=dim;
    pimaginary2+=dim;
  }

  /* now back into the original domain, then shift the array for simplicity */
  realfft2d_in_place(real,bits,i,wr,wl);
  shift_array(real,dim);
  /* temporary display results
  //return(1);

  /* then find the top 'candidate' number of points, loading their parameters along the way */
  for(i=0;i<number_candidates;i++){
    highest = -(float)1e20;
    for(j=0;j<dim;j++){
      preal1 = real1;
      pimaginary1 = real1[dim];
      for(k=0;k<dim;k++){
        if( *preal1 > highest ){
          /* check to see if this is within PICK_RADIUS of a previous choice */
          ok = 1;
          l = i;
          while( l-- > 0 ){
            if( abs(j-y_off[l]) < PICK_RADIUS ||
              abs(j-dim-y_off[l]) < PICK_RADIUS ||
              abs(j-dim-y_off[l]) < PICK_RADIUS ||
              if( abs(k-x_off[l]) < PICK_RADIUS ||
                abs(k+dim-x_off[l]) < PICK_RADIUS ||
                abs(k-dim-x_off[l]) < PICK_RADIUS )ok=0;
            }
          }
          if(ok){

```

```

    highest = *preal1;
    x_off[l] = k;
    y_off[l] = j;
  }
  }
  preal1++;
}

/* step through the found candidates, finding inter-sample values for the peak location */
for(i=0;i<number_candidates;i++){
  ymedian[0]=ymedian[1]=ymedian[2]=(float)0.0;
  xmedian[0]=xmedian[1]=xmedian[2]=(float)0.0;
  py = ymedian;
  for(j=-1;j<2;j++){
    jtemp = y_off[i]+j;
    if(jtemp < 0)jtemp=dim-1;
    else if(jtemp==dim)jtemp=0;
    px = xmedian;
    for(k=-1;k<2;k++){
      ktemp = x_off[i]+k;
      if(ktemp < 0)ktemp=dim-1;
      else if(ktemp==dim)ktemp=0;
      *py += real1[jtemp*dim+ktemp];
      *px += real1[jtemp*dim+ktemp];
    }
    py++;
  }
  /* now find median values */
  ratio = get_median_float(ymedian);
  y_offset[i] = (float)dim2 - ( (float)y_off[i] + ratio );
  ratio = get_median_float(xmedian);
  x_offset[i] = (float)dim2 - ( (float)x_off[i] + ratio );
  value[i] = real1[x_off[i] + dim*y_off[i]];
}

return(1);
}

/* simple sub-routine for direct_registration
int get_working_dimension(
  int alignment_mode,
  int xdim1,
  int ydim1,
  int xdim2,
  int ydim2,
  int *downsample
){
  int highest=xdim1,go=1,ftdim;

  if(ydim1>highest)highest = ydim1;
  if(xdim2>highest)highest = xdim2;
  if(ydim2>highest)highest = ydim2;

  switch(alignment_mode){
    case 0: /* no downsampling
      *downsample = 1;
      ftdim = 1;
      while( go ){
        if( highest > ftdim ){
          ftdim*=2;
        }
        else go = 0;
      }
      break;
    case 1: /* nominal downsampling
      *downsample = ((highest-1)/NOMINAL_DOWNSAMPLE_DIM)+1;
      ftdim = NOMINAL_DOWNSAMPLE_DIM;
      break;
    case 2: /* super downsampling
      *downsample = ((highest-1)/SUPER_DOWNSAMPLE_DIM)+1;
      ftdim = SUPER_DOWNSAMPLE_DIM;
      break;
  }

  return(ftdim);
}

/* another sub-routine for direct_registration
int copy_downsample_window(
  unsigned char *in,
  int xdim,
  int ydim,
  float *out,
  int outdim,
  int outsample
){
  unsigned char *pin;
  int i,j;

```



```

float *pout, *pwindow, normalize;
pin = in;
memset(out, 0, outdim*outdim*sizeof(float));
for(i=0; i<ydim; i++){
    pout = &out[i*outdim];
    for(j=0; j<xdim; j++){
        pout[j/downsample] += (float)*(pin++);
    }
}

// normalize it for downsampling
if(downsampling > 1){
    xdim = 1 + (xdim-1)/downsample;
    ydim = 1 + (ydim-1)/downsample;
    normalize = (float)downsample * (float)downsample;
    for(i=0; i<ydim; i++){
        pout = &out[i*outdim];
        for(j=0; j<xdim; j++){
            *(pout++) /= normalize;
        }
    }
}

if(WINDOW ORIGINALS){
    float *window_function = new float(outdim);
    load_windowing_function(xdim, window_function);
    pout = out;
    for(i=0; i<ydim; i++){
        pwindow = &window_function[i];
        for(j=0; j<xdim; j++){
            *(pout++) += *(pwindow++);
        }
        pout += (outdim-xdim);
    }
    load_windowing_function(ydim, window_function);
    pout = out;
    for(i=0; i<ydim; i++){
        pwindow = &window_function[i];
        for(j=0; j<xdim; j++){
            *(pout++) += *(pwindow);
        }
        delete () window_function;
    }
    return(1);
}

int fourier_mellin_transform(
    float *in,
    float *ftemp,
    int dim,
    float *out
){
    int i, j;
    float *pout, *pwindow;

    convert_to_magnitude(ftemp, in, dim);
    log_polar_map(ftemp, out, dim);
    if(WINDOW LOGPOLAR_MAP){
        load_window_function = new float(lp_sampling);
        load_windowing_function(lp_sampling, window_function);
        pout = out;
        for(i=0; i<lp_sampling; i++){
            pwindow = &window_function[i];
            for(j=0; j<lp_sampling; j++){
                *(pout++) += *(pwindow);
            }
            delete () window_function;
        }
        return(1);
    }

    int get_best_candidate(
        int number_candidates,
        float *ftemp,
        int dim,
        float *in,
        int xdim,
        int ydim,
        int xdim_orig,
        int ydim_orig,
        int downsample,
        float *rotation,
        float *scale,
        float *x_trans,
        float *y_trans,
        float *template_real
    ){
        int i, highest_i, j;
        float highest = -(float)1e20, xtrans, ytrans, value;

        for(i=0; i<number_candidates; i++){
            for(j=0; j<2; j++){
                /* rotate and scale suspect real image into ftemp */
                rotate_scale_translate_image(ftemp, dim, in, xdim, ydim, xdim_orig, ydim_orig,
                    downsample, rotation[i], (float)j*(float)180.0, scale[i]);
                realfft2_in_place(ftemp, bits, 0, wr, wl);
                gmft(template_real, ftemp, dim, bits, 1, xtrans, ytrans, &value, 1);
                if(value > highest){
                    highest = value;
                    highest_i = i;
                    if(j==1) rotation[i] += (float)180.0;
                    x_trans[i] = xtrans;
                    y_trans[i] = ytrans;
                }
            }
        }
        rotation[0] = rotation[highest_i];
        scale[0] = scale[highest_i];
        x_trans[0] = x_trans[highest_i];
        y_trans[0] = y_trans[highest_i];
        return(1);
    }

    double log_id_remap(
        float *in,
        float *out,
        int dim
    ){
        int i, dim2 = dim/2, xx;
        float *pin, *pout;
        double radius, frack;
        double scale_increment_id;

        scale_increment_id = pow( 1.0/(double)START_RADIUS_ID, 1.0/(double)dim);
        pout = out;
        for(i=0; i<dim; i++){
            radius = (START_RADIUS_ID*(double)dim2) * pow(scale_increment_id, (double)i);
            xx = (int)radius;
            frack = radius - (double)xx;
            pin = &in[xx];
            *pout = (float) ( (1.0-frack) * (double)*(pin++) );
            *(pout++) += (float) ( frack * (double)*pin );
        }
        return(scale_increment_id);
    }

    int gmft_id(
        float *real1,
        float *imaginary1,
        float *real2,
        float *imaginary2,
        int dim,
        int bits,
        float *offset
    ){
        int i, highest_i;
        float *preall, *preal2, *pimaginary1, *pimaginary2;
        float mag1, mag2, dot, dott, cross, median[3], highest, ratio, ftmp;

        /* calculate phase differences and reload them into real1 and imaginary1 */
        /* keep phase differences to pi to -pi */
        preall = real1, pimaginary1 = imaginary1;
        preal2 = real2, pimaginary2 = imaginary2;
        for(i=0; i<dim; i++){
            mag1 = (float)sqrt( (double)(*(preall) * *(preall) + *(pimaginary1) * *(pimaginary1)) );
            mag2 = (float)sqrt( (double)(*(preal2) * *(preal2) + *(pimaginary2) * *(pimaginary2)) );
            if(mag1 == (float)0.0) mag1 = (float)SMALL;
            if(mag2 == (float)0.0) mag2 = (float)SMALL;
            dott = *(preall) * *(preal2) + *(pimaginary1) * *(pimaginary2);
            if(dott < (float)0.0) dott = (float)0.0;
            dott = (float)sqrt( (double)dott );
            cross = *(preall) * *(pimaginary2) - *(preal2) * *(pimaginary1);
            if(cross < (float)0.0) cross = -(float)1.0;
            else cross = (float)1.0;
            ftmp = mag2;
            dott = ftmp/dott + ftmp;
            *(preall++) = dott;
        }
    }
}

```

```

    * (pimaginary1++) = cross*dott;
}

fft(reall,imaginary1,bits,1,wr,w1,1);

/* search for highest value, then median find the center */
preall = reall;
for(i=0;i<dim;i++){
    if( preall > highest){
        highest = preall;
        highest_i = i;
    }
    preall++;
}

if(highest_i == 0){
    median[0] = reall[0];
    median[1] = reall[1];
    median[2] = reall[2];
}
else if(highest_i == (dim-1)){
    median[0] = reall[dim-1];
    median[1] = reall[dim-1];
    median[2] = reall[dim-1];
}
else {
    median[0] = reall[highest_i-1];
    median[1] = reall[highest_i+1];
    median[2] = reall[highest_i];
}

ratio = get median float(median);
*offset = (float)highest_i / ratio;
if (*offset > (float)dim/2.0) *offset -= (float)dim;
return(1);
}

int refine_axis(
    unsigned char *template,
    int template_xdim,
    int template_ydim,
    unsigned char *suspect,
    int suspect_xdim,
    int suspect_ydim,
    float *x,
    float *y,
    int which
){
    unsigned char *psuspect;
    int i,j,highest,fftdim,bits,xx,yy,xdim,ydim;
    float x0,x1,x2,y0,y1,y2,psuspect_integral,*template_integral;
    float scan_x,scan_y,jump_x,jump_y,current_x,current_y;
    float scale,translation,distance,ydistance,suspect_dc,template_dc,frac;
    double scale_increment_id;

    /* first convert the y axis version to the x axis version */
    x0 = x[0]; y0 = y[0];
    if(which){
        x1 = x[2]; y1 = y[2];
        x2 = x[1]; y2 = y[1];
        xdim = suspect_xdim;
        ydim = suspect_ydim;
    }
    else {
        x1 = x[1]; y1 = y[1];
        x2 = x[2]; y2 = y[2];
        xdim = suspect_xdim;
        ydim = suspect_ydim;
    }

    /* determine the next highest power of two above higher of the two suspect axes */
    if(suspect_xdim > suspect_ydim) highest = suspect_xdim;
    else highest = suspect_ydim;
    bits = 1 + (int)( log( (double)highest - 0.5 ) / log(2.0) );
    fftdim = (int)pow(2.0,(double)bits + 0.00000001);

    float *template_integral = new float[fftdim];
    float *suspect_integral = new float[fftdim];
    float *template_integral_imaginary = new float[fftdim];
    float *suspect_integral_imaginary = new float[fftdim];
    float *template_integral_copy = new float[fftdim];
    float *suspect_integral_copy = new float[fftdim];

    /* load suspect integral waveform */
    psuspect_integral = suspect_integral;
    for(i=0;i<fftdim;i++){
        if(which){
            psuspect = suspect;

```

```

convert to magnitude,ld inplace(suspect_integral,suspect_integral_imaginary,fftldim);
convert to magnitude,ld inplace(template_integral,template_integral_imaginary,fftldim);
// next routine places output into suspect_integral_imaginary array
scale_increment_ld = log10(remap(suspect_integral,suspect_integral_imaginary,fftldim);
scale_increment_ld = log10(remap(template_integral,template_integral_imaginary,fftldim);
// copy output back into function array and zero out imaginary
memcpy(suspect_integral,template_integral,sizeof(float)*fftldim);
memset(suspect_integral_imaginary,0,sizeof(float)*fftldim);
memset(template_integral,template_integral_imaginary,sizeof(float)*fftldim);
memset(template_integral_imaginary,0,sizeof(float)*fftldim);
// now do the 1d fourier transform
window_id_vector(template_integral,fftldim,fftldim);
fft(template_integral,suspect_integral,fftldim,fftldim,bits,0,wr,wl,1);
fft(template_integral,template_integral_imaginary,bits,0,wr,wl,1);
/* gmf ld to find any small scaling difference between the two */
gmf_ld(suspect_integral,suspect_integral_imaginary,template_integral,
template_integral_imaginary,fftldim,bits,acscale);
//scale *= (float)0.6; // slight damping factor
scale = (float)pow(scale_increment_ld,(double)scale);
// update the x's and y's
xdistance = (x1-x0);
ydistance = ((float)1.0 - scale);
ydistance = (y1-y0);
x[3] += xdistance; y[3] += ydistance;
x[4] += xdistance/(float)2.0; y[4] += ydistance/(float)2.0;
if (which) {
x[2] += xdistance; y[2] += ydistance;
x1 = x[2]; y1 = y[2];
} else {
x[1] += xdistance; y[1] += ydistance;
x1 = x[1]; y1 = y[1];
}
}
// now with the new scale information, perform a gmf on the original and its rescaled
countercat++;
template_integral = template_integral;
scale_ld(fftldim,1.0 / scale);
for(i=0;current_x<(float)0.0;i<xdim,i++,current_x+=scale){
if (ix >= xdim-1){ptemplate_integral++} = lllast;
else {
frac = current_x - (float)xx;
template_integral = ((float)1.0 - frac) * template_integral_copy[xx];
template_integral += frac * template_integral_copy[xx+1];
}
lllast = *(ptemplate_integral-1);
}
// window the new scaled array; other one should be copy of windowed original
memcpy(suspect_integral,suspect_integral_copy,sizeof(float)*fftldim);
window_id_vector(template_integral,xdim,fftldim);
window_id_vector(suspect_integral,xdim,fftldim);
memset(template_integral_imaginary,0,sizeof(float)*fftldim);
memset(template_integral_imaginary,0,sizeof(float)*fftldim);
fft(suspect_integral,suspect_integral_imaginary,bits,0,wr,wl,1);
fft(template_integral,template_integral_imaginary,bits,0,wr,wl,1);
// now find the translation
gmf_ld(suspect_integral,suspect_integral_imaginary,template_integral,
template_integral_imaginary,fftldim,bits,translation);
// adjust x and y accordingly
translation *= (float)0.5; // I think this accounts for the fact that scaling has changed
origins????? very kludge
scan_x = translation;
scan_y = translation;
x[0] += scan_x; y[0] += scan_y;
x[1] += scan_x; y[1] += scan_y;
x[2] += scan_x; y[2] += scan_y;
x[3] += scan_x; y[3] += scan_y;
x[4] += scan_x; y[4] += scan_y;
delete [] template_integral;
delete [] suspect_integral;
delete [] template_integral_imaginary;
delete [] suspect_integral_imaginary;
delete [] template_integral_copy;
delete [] suspect_integral_copy;
return(0);
}
float refined_rotation(

```

```

float *x,
float *y,
unsigned char *suspect,
int suspect_xdim,
int suspect_ydim,
unsigned char *template,
int template_xdim,
int template_ydim
){
int i,xx,yy,count_template,count_suspect;
float line_integral[REFINED_ROTATION_DIMENSION],*pli,*pli_template;
float line_integral[REFINED_ROTATION_DIMENSION];
float line_integral_imaginary[REFINED_ROTATION_DIMENSION];
float angle,x_suspect,y_suspect,x1_suspect,y1_suspect,dx_suspect,dy_suspect;
float x_template,y_template,x1_template,y1_template,dx_template,dy_template;
float top_x_suspect=(float)(suspect_xdim-1),top_y_suspect=(float)(suspect_ydim-1);
float a_const,b_const,tweak,dc_suspect,dc_template;
float new_x,new_y,axis_x,axis_y;
axis_x = (x[2]-x[0])/(float)(suspect_ydim-1); // this gives the unit vector in terms of
the suspect array */
axis_y = (y[2]-y[0])/(float)(suspect_ydim-1);
axis_x = (x[1]-x[0])/(float)(suspect_xdim-1);
axis_y = (y[1]-y[0])/(float)(suspect_xdim-1);
/* create line integral sweep around suspect's and template's center point */
pli = line_integral;
pli_template = line_integral;
dc_suspect = dc_template=(float)0.0;
for(i=0;i<REFINED_ROTATION_DIMENSION,i++){
angle = (float)i * (float)PI / (float)REFINED_ROTATION_DIMENSION;
x_suspect = x1_suspect = (float)0.5 + top_x_suspect/(float)2.0;
y_suspect = y1_suspect = (float)0.5 + top_y_suspect/(float)2.0;
dx_suspect = (float)sin((double)angle);
dy_suspect = (float)cos((double)angle);
x_suspect+=dx_suspect;x1_suspect+=dx_suspect;
y_suspect+=dy_suspect;y1_suspect+=dy_suspect;
x_template = x1_template = (float)0.5*x[4];
y_template = y1_template = (float)0.5*y[4];
dx_template = (axis_x+dx_suspect,axis_x+dy_suspect);
dy_template = (axis_y+dx_suspect,axis_y+dy_suspect);
x_template+=dx_template;x1_template+=dx_template;
y_template+=dy_template;y1_template+=dy_template;
*pli = (float)0.0;
*pli_template = (float)0.0;
count_template=0;count_suspect=0;
while(x_suspect>0.0 && x_suspect<top_x_suspect && y_suspect>0.0 &&
y_suspect<top_y_suspect){
xx = (int)x_suspect;
yy = (int)y_suspect;
*pli += suspect[yy*suspect_xdim+xx];
xx = (int)x1_suspect;
yy = (int)y1_suspect;
*pli += suspect[y1*suspect_xdim+xx];
x_suspect+=dx_suspect;x1_suspect+=dx_suspect;
y_suspect+=dy_suspect;y1_suspect+=dy_suspect;
count_suspect++;
}
if(y_template>0.0&&y_template<top_y_template&&x_template>0.0&&x_template<top_x_template){
&y1_template=0.0&y1_template<top_y_template&&x1_template>0.0&&x1_template<top_x_template){
xx = (int)x_template;
yy = (int)y_template;
*pli_template += ttemplate[yy*template_xdim+xx];
xx = (int)x1_template;
yy = (int)y1_template;
*pli_template += ttemplate[y1*template_xdim+xx];
x_template+=dx_template;x1_template+=dx_template;
y_template+=dy_template;y1_template+=dy_template;
count_template++;
}
}
*pli /= (float)count_suspect;
*pli_template /= (float)count_template;
dc_suspect += *pli++;
dc_template += *pli_template++;
}
/* now one-d fft them and one d gmf */
memset(line_integral_imaginary,0,sizeof(float)*REFINED_ROTATION_DIMENSION);

```

```

memset(line_integral_template_imaginary,0,sizeof(float))*REFINED_ROTATION_DIMENSION);
pli = line_integral;
pli_template = line_integral_template;
dc_suspect /= (float)REFINED_ROTATION_DIMENSION;
dc_template /= (float)REFINED_ROTATION_DIMENSION;
for(i=0;i<REFINED_ROTATION_DIMENSION;i++){
    *pli++ += dc_suspect;
    *pli_template++ += dc_template;
}
fft(line_integral,line_integral_imaginary,REFINED_ROTATION_BITS,0,w1,1);
fft(line_integral_template,line_integral_template_imaginary,REFINED_ROTATION_BITS,0,w1,1);
gsmf_id(line_integral,line_integral_imaginary,line_integral_template,line_integral_template_imaginary,
    REFINED_ROTATION_DIMENSION,REFINED_ROTATION_BITS,tlewak);
tlewak *= (float)0.5; // slight damping factor

tlewak *= -((float)180.0/(float)REFINED_ROTATION_DIMENSION);
/* update xy0 thru xy3 */
a_const = (float)cos( (double)tlewak * PI /180.0 );
b_const = (float)sin( (double)tlewak * PI /180.0 );
new_x = a_const*(x[4]-x[0]) - b_const*(y[4]-y[0]);
new_y = b_const*(x[4]-x[0]) + a_const*(y[4]-y[0]);
x[0] = x[4] - new_x;
y[0] = y[4] - new_y;
new_x = a_const*(x[4]-x[1]) - b_const*(y[4]-y[1]);
new_y = b_const*(x[4]-x[1]) + a_const*(y[4]-y[1]);
x[1] = x[4] - new_x;
y[1] = y[4] - new_y;
new_x = a_const*(x[4]-x[2]) - b_const*(y[4]-y[2]);
new_y = b_const*(x[4]-x[2]) + a_const*(y[4]-y[2]);
x[2] = x[4] - new_x;
y[2] = y[4] - new_y;
new_x = a_const*(x[4]-x[3]) - b_const*(y[4]-y[3]);
new_y = b_const*(x[4]-x[3]) + a_const*(y[4]-y[3]);
x[3] = x[4] - new_x;
y[3] = y[4] - new_y;
return(tlewak);
}

int Align::fine_tune_x_y(unsigned char *template,
    int template_xdim,
    int template_ydim,
    unsigned char *suspect,
    int suspect_xdim,
    int suspect_ydim,
    float *x,
    float *y,
    float *rotation)
{
    //int foo=1;
    float refinement;
    //while(foo){
    // find scale, xtrans optimal pair */
    refine_axis(template_xdim,template_ydim,suspect,suspect_xdim,
        suspect_ydim,x,y,0);
    // find yscale, ytrans optimal pair */
    refine_axis(template_xdim,template_ydim,suspect,suspect_xdim,
        suspect_ydim,x,y,1);
    // fine tune rotation */
    refinement = refined_rotation(x,y,suspect,suspect_xdim,suspect_ydim,template,
        template_xdim,template_ydim);
    // NOTE: SOME CONFUSION ABOUT WHETHER NEXT LINE SHOULD BE == OR +=
    // rotation += refinement;
    //}
    m_alignStatus.refinement = refinement;
    return(1);
}

/* subroutine for direct registration */
int get_corners_and_center(
    float *x,
    float *y,
    float rotation,
    float scale,
    float x_trans,
    float y_trans,
    int xdim,
    int ydim,
    int fftdim,
    int downsample)
{

```

```

float a_const,b_const;
/* the center of the suspect array should translate to...
(fftdim*downsample - 1)/2.0 - x_trans*downsample, same on y??? */
/* note that the origin of the downsampled arrays actually is
positioned at (downsample-1)/2, (downsample-1)/2 in the coordinates of the
original arrays */
x_trans *= (float)downsample;
y_trans *= (float)downsample;
x[4] = (float)(fftdim*downsample - 1)/(float)2.0 + x_trans;
y[4] = (float)(fftdim*downsample - 1)/(float)2.0 + y_trans;
a_const = (float)cos((double)rotation*PI/180.0)/scale;
b_const = (float)sin((double)rotation*PI/180.0)/scale;
x[0] = x[4] - (a_const*(float)(xdim-1) - b_const*(float)(ydim-1))/(float)2.0;
y[0] = y[4] - (b_const*(float)(xdim-1) + a_const*(float)(ydim-1))/(float)2.0;
x[1] = x[4] + (a_const*(float)(xdim-1) + b_const*(float)(ydim-1))/(float)2.0;
y[1] = y[4] + (b_const*(float)(xdim-1) - a_const*(float)(ydim-1))/(float)2.0;
x[2] = x[4] - (a_const*(float)(xdim-1) + b_const*(float)(ydim-1))/(float)2.0;
y[2] = y[4] - (b_const*(float)(xdim-1) - a_const*(float)(ydim-1))/(float)2.0;
x[3] = x[4] + (a_const*(float)(xdim-1) - b_const*(float)(ydim-1))/(float)2.0;
y[3] = y[4] + (b_const*(float)(xdim-1) + a_const*(float)(ydim-1))/(float)2.0;
return(1);
}

int final_image(
    unsigned char *out,
    int outxdim,
    int outydim,
    unsigned char *in,
    int inxdim,
    int inydim,
    float *x,
    float *y,
    int num_channels,
    int option)
{
    unsigned char *pout;
    int i,j,xx,yy;
    float ii,current_x,current_y,fracy,fracy,tmp,tmp1,tmp2,tmp3,tmp4;
    float x_start,y_start,scan_x,scan_y,jump_x,jump_y;
    unsigned char *pin;
    if(option == 1){ // clear ttemplate array
        pout=out;
        for(ii=0;i<(num_channels*outxdim*outydim);i++){*(pout++)=(unsigned char)0;
        }
        yaxis_x = (x[2]-x[0])/(float)(inydim-1); // this gives the unit vector in terms of the
        suspect array */
        yaxis_y = (y[2]-y[0])/(float)(inydim-1);
        yaxis_dist = (float)sqrt((double)(yaxis_x*yaxis_x+yaxis_y*yaxis_y));
        xaxis_x = (x[1]-x[0])/(float)(inxdim-1);
        xaxis_y = (y[1]-y[0])/(float)(inxdim-1);
        xaxis_dist = (float)sqrt((double)(xaxis_x*xaxis_x+xaxis_y*yaxis_y));
        /* starts is origin dotted with axes */
        x_start = (-x[0]*xaxis_x - y[0]*xaxis_y)/xaxis_dist/xaxis_dist;
        y_start = (-x[0]*xaxis_x - y[0]*xaxis_y)/yaxis_dist/yaxis_dist;
        scan_x = xaxis_x/xaxis_dist/xaxis_dist;
        scan_y = yaxis_y/yaxis_dist/yaxis_dist;
        jump_x = xaxis_y/yaxis_dist/xaxis_dist;
        jump_y = yaxis_x/yaxis_dist/yaxis_dist;
        pout = out;
        for(ii=0;i<outydim;i++){
            ii = (float)1;
            current_x = x_start + ii * jump_x;
            current_y = y_start + ii * jump_y;
            if(num_channels==1){
                for(j=0;j<outxdim;j++){
                    if(current_x<(float)0.0 || current_x>(float)(inxdim-1) || current_y<(float)0.0
                        || current_y>(float)(inydim-1)){
                        if(option == 0)pout++; // this option preserves the rest of template
                        else *(pout++) = (unsigned char)0;
                    }
                    else {
                        xx = (int)current_x;
                        yy = (int)current_y;
                        fracy = current_x - (float)xx;
                        fraxy = current_y - (float)yy;
                    }
                }
            }
        }
    }
}

```

```

pin = sin[yy*inxdim + xx];
ftmp = ((float)1.0 - frack) * ((float)1.0 - frack) * (float)*(pin++);
frack = ((float)1.0 - frack) * (float)*(pin);
pin += (inxdim-1);
ftmp += ((float)1.0 - frack) * frack * (float)*(pin++);
frack = ((float)1.0 - frack) * (float)*(pin);
// debug lines, use with option -o, then it draws a dashed line around suspect
if (xx == 0 || yy == (inxdim-2) || yy == 0 || yy == (inydim-2)) { (pout++) = (unsigned char)0;
else { (pout++) = (unsigned char)ftmp;
}
}
current_x += scan_x;
current_y += scan_y;
}
}
else if (num_channels == 3) {
for (j=0; j<outxdim; j++) {
if (current_x < (float)0.0 || current_x > (float)(inxdim-1) || current_y < (float)0.0
|| current_y > (float)(inydim-1)) {
if (option == 0) { pout += 3; // this option preserves the rest of template
else { (pout++) = *(pout++) = *(pout++) = *(pout++) = (unsigned char)0;
}
}
else {
xx = (int)current_x;
yy = (int)current_y;
frack = current_x - (float)xx;
fracy = current_y - (float)yy;
ftmp1 = ((float)1.0 - frack) * ((float)1.0 - frack);
ftmp2 = frack * ((float)1.0 - frack);
ftmp3 = ((float)1.0 - frack) * frack;
ftmp4 = frack * frack;
pin = sin[3*(yy*inxdim + xx)];
ftmp = ftmp1 * (float)*pin;
pin += 3;
ftmp += (ftmp2 * (float)*pin);
pin += 3*(inxdim-1);
pin += 3*(ftmp3 * (float)*pin);
pin += 3;
ftmp += (ftmp4 * (float)*pin);
*(pout++) = (unsigned char)ftmp;
pin = sin[3*(yy*inxdim + xx)+1];
ftmp = ftmp1 * (float)*pin;
pin += 3;
ftmp += (ftmp2 * (float)*pin);
pin += 3*(inxdim-1);
pin += 3*(ftmp3 * (float)*pin);
pin += 3;
ftmp += (ftmp4 * (float)*pin);
*(pout++) = (unsigned char)ftmp;
pin = sin[3*(yy*inxdim + xx)+2];
ftmp = ftmp1 * (float)*pin;
pin += 3;
ftmp += (ftmp2 * (float)*pin);
pin += 3*(inxdim-1);
pin += 3*(ftmp3 * (float)*pin);
pin += 3;
ftmp += (ftmp4 * (float)*pin);
*(pout++) = (unsigned char)ftmp;
}
current_x += scan_x;
current_y += scan_y;
}
}
return(1);
}

// new public subliminal grid stuff, Late April 1996
float *subliminal_grid = new float[inx*128];
float *mellin_transform = new float[inx*128];
int grid_freq_total = 16;
float *grid_phase = new float[grid_freq_total];
int *grid_x = new int[grid_freq_total];
int *grid_y = new int[grid_freq_total];

int log_polar_remap_public(
float *in,
float *out,
int dim)
{
int i, dim2 = dim/2, xx, yy, j;
float *pin, *pout;
double cheta, dx, dy, radius [MELLIN_DIMENSION], x, y, frack, fracy, *pradius;
int n = MELLIN_DIMENSION;
double increment = pow( 2.0 , 0.025 );

```

```

double start = sqrt(32.5);
for (i=0; i<n; i++) {
radius[i] = start * pow(increment, (double)i);
}
// pre-filter fourier mag data;
// first add 90 degree separated
for (i=0; i<64; i++) {
for (j=0; j<64; j++) { // output into left half of original array
in[63-i*128+j] += in[(i+1)*128+64+j];
}
}
float local_average, *p1, *p2, *p3;
for (i=0; i<64; i++) {
pout = in[64*128+i]; // output into right half of original array
if (i==0) p1 = in;
else p1 = in[(i-1)*128];
p2 = in[(i-1)*128];
if (i==63) p3 = in[63*128];
else p3 = in[(i+1)*128];
// first element
local_average = *(p1 + *(p1+1) + *(p2+1) + *(p3+1) + *p3) / (float)5.0;
if (*p2 > (float)100.0 * local_average) {
*pout = (float)100.0;
}
else if (local_average < SMALL) {
*pout = SMALL;
}
else {
*pout = *p2 / local_average;
}
}
p1 += p2++; p3++;
pout += 128;
for (j=1; j<63; j++) {
local_average = *(p1-1) + *p1 + *(p1+1) + *(p2+1);
local_average /= (float)8.0;
if (*p2 > (float)100.0 * local_average) *pout = (float)100.0;
else if (*p2 < SMALL) *pout = SMALL;
else *pout = *p2 / local_average;
p1 += p2++; p3++;
pout += 128;
}
// last element
local_average = *(p1 + *(p1-1) + *(p2-1) + *(p3-1) + *p3) / (float)5.0;
if (*p2 > (float)100.0 * local_average) *pout = (float)100.0;
else if (*p2 < SMALL) *pout = SMALL;
else *pout = *p2 / local_average;
}
// copy horizontal row into vertical column for interp porpoises
for (i=1; i<64; i++) in[64+i] = in[64+i*128];
pout = out;
for (theta=0.0; j<n; j++, theta += (PI/((double)n)/2.0)) {
dx = cos(theta);
dy = sin(theta);
pradius = radius;
pout = &out[j];
for (i=0; i<n; i++) {
x = (double)dim2 + *pradius * dx;
y = *(pradius++) * dy;
xx = (int)x;
yy = (int)y;
frack = x - (double)xx;
fracy = y - (double)yy;
pin = sin[yy*dim + xx];
*pout = (float) ( (1.0 - frack) * (1.0 - fracy) * (double)*(pin++) );
*pout += (float) ( frack * (1.0 - fracy) * (double)*pin );
pin += (dim-1);
*pout += (float) ( (1.0 - frack) * fracy * (double)*(pin++) );
*pout += (float) ( frack * fracy * (double)*pin );
pout += n;
}
}
return(1);
}

load_grid_family(
) {
static int done = 0;
// don't change this without checking its effects on the later grid finding routines
// such as resolve_orientation

```

```

done = 0; // force it for now
if(!done){
    delete [] wr;
    delete [] wi;
    delete [] mag_buffer;
    done = 1;
}
return(1);
}

// specific to hunt_for_grid
int add_block_magnitude(
    unsigned char *data,
    float *fourier_mag,
    int n, // power of 2 dimension of fourier mag
    float *buffer, // needs to be n*(n+2) in length
    int xbumps,
    int ybumps,
    int bump_size,
    int xdim,
    int original_xdim, // pixel based jump pointer for moving down rows
    int truncated
){
    // load fourier array with bump data
    unsigned char *pdata = data;
    float *pbuffer;
    for(i=0; i<(n*(n+2)); i++){
        pbuffer = &buffer[i*n];
        load_bump_array(
            pbuffer, // floating point bump array to be filled (output)
            pdata, // input pixel data
            xbumps, // number of bumps in this row (not pixels)
            xdim, // number of channels
            bump_size, // pixels per bump
            original_xdim - xbumps*bump_size, // number of raw pixels between
            (xdim*bump_size) // do not overflow the bump buffer
        );
        pdata += (xdim*original_xdim*bump_size);
    }
    // window it if you please
    float *window_function = new float[128];
    load_window_function(128, window_function);
    float *pwindow_row = window_function;
    float *pwindow_column = window_function;
    pbuffer = buffer;
    for(i=0; i<128; i++){
        pwindow_column = window_function;
        for(j=0; j<128; j++){
            *pbuffer++ = *pwindow_row * *pwindow_column++;
        }
        pwindow_row++;
    }
    delete [] window_function;
    // // this doesn't seem to help at all! results seem to get worse

    // fft the dog
    int bits = (int) (log( (double)(n+1) ) / log( 2.0 ) ); // fftdim should always be power
    of 2
    realfft2d_in_place(buffer, bits, 0, wr, wi);

    // now add its magnitude into the accumulator array
    float *pfourier = fourier_mag;
    float *preal = buffer;
    float *pimag = &buffer[n];
    for(i=0; i<(n/2+1); i++){
        for(j=0; j<n; j++){
            // consider a "cheaty" version of the following, just add the absolute mags,
            *pfourier++ = (float)sqrt(*preal * *preal + *pimag * *pimag);
            preal++; pimag++; pfourier++;
        }
        preal += n;
        pimag += n;
    }
    return(1);
}

int rotate_scale_image(
    unsigned char *data,

```

```

/* this is a specialized function simply meant to find out which of 4
90 degree orientations is the true orientation of the subliminal grid;
the Fourier mellen transform, combined with our "roiding" of frequencies,
gives this ambiguity in the first place
*/
int resolve_orientation(
    unsigned char *data,
    int xdim,
    int ydim,
    int zdim,
    int bump_size,
    int n, // power of 2 used in inverse fft's
    int original_xdim,
    float *rotation,
    float *scale
) {
    int mult = 1;
    if (*scale > (float)1.25) { // up n to the next higher power of two
        n = 2;
        mult = 2;
    }

    float *buffer = new float[n*(n+2)];
    int n2 = n/2+1;

    rotate_scale_image(
        data,
        xdim,
        ydim,
        zdim,
        bump_size,
        n,
        original_xdim,
        *rotation,
        *scale,
        buffer
    );

    // fft the thing
    int bits = (int)(log( (double)(n+1) ) / log( 2.0 ) ); // fftdim should always be power
    of 2
    realfft2d_in_place(buffer, bits, 0, wr, w1); // ultimately, direct calculation may be faster
    assuming frequency points < bits*bits

    // save the original phase values
    float *real = new float[grid_freq_total];
    float *imag = new float[grid_freq_total];
    for (i=0; i<grid_freq_total; i++) {
        real[i] = buffer[n2 + mult*grid_x(i) + 2*n*mult*grid_y(i)];
        imag[i] = -buffer[n2 + mult*grid_x(i) + 2*n*mult*grid_y(i)];
    }

    // now step through the four possible orientations, finding the best fit
    // the current location of this routine is intimately tied to
    // the function load_grid_family
    float highest_high = (float)-1e20;
    int high1_tmp;
    float value[4];
    for (i=0; i<4; i++) {
        // zero out buffer
        memset(buffer, 0, sizeof(float)*n*(n+2));
        // multiply this orientation by saved phases
        for (j=0; j<grid_freq_total; j++) {
            if (i==0) {
                grid_real = (float)cos((double)grid_phase[j]);
                grid_imag = (float)sin((double)grid_phase[j]);
            }
            else if (i==1) {
                tmp = (j+grid_freq_total/2)%grid_freq_total;
                grid_real = (float)cos((double)grid_phase[tmp]);
                if (tmp >= grid_freq_total/2) grid_imag = (float)sin((double)grid_phase[tmp]);
                else grid_imag = -(float)sin((double)grid_phase[tmp]);
            }
            else if (i==2) {
                grid_real = (float)cos((double)grid_phase[j]);
                grid_imag = -(float)sin((double)grid_phase[j]);
            }
            else {
                tmp = (j+grid_freq_total/2)%grid_freq_total;
                grid_real = (float)cos((double)grid_phase[tmp]);
                if (tmp >= grid_freq_total/2) grid_imag = -(float)sin((double)grid_phase[tmp]);
                else grid_imag = (float)sin((double)grid_phase[tmp]);
            }
        }
        buffer[n2 + mult*grid_x(j) + 2*n*mult*grid_y(j)] = real(j) * grid_real -
        imag(j) * grid_imag;
        buffer[n + n2 + mult*grid_x(j) + 2*n*mult*grid_y(j)] = real(j) * grid_imag +
        imag(j) * grid_real;
    }
}

```

```

}
realfft2d_in_place(buffer,bits,1,wr,w1); // ultimately, direct calculation may be faster
assuming frequency points < bits*bits

// find highest point
highest = (float)-1e20;
float *buffer = buffer;
int high_x,high_y;
for(j=0;j<(n*n);j++){
    if(*buffer > highest){
        highest = *buffer;
        high_x = j/n;
        high_y = j - high_x;
    }
    buffer++;
}

// load its median inter-sample value
value[i] = get_2d_median(buffer,n,n,high_x,high_y,&x_offset[i],&y_offset[i]);

// then, find the highest of the four
if(highest > high){
    high = i;
    high = highest;
}

// update rotation
*rotation += (float)highi * (float)90.0;

delete [] real;
delete [] imag;

return(i);
}

/*
This function performs two basic services, first, it simply attempts
to determine if a public subliminal grid exists or not;
if one does exist then the second basic service is to determine the
rough scale and rotation state of that grid.

The mode_flag variable provides options for how fast v. thorough the algorithms
are.
*/

int hunt_for_grid(
    unsigned char *data, // input image, unknown signature status
    int xdim, // its full pixel dimension in x
    int ydim, // ditto in y
    int zdim, // number of channels
    int probable_bump_size, // this is a tricky one to start; to best function,
    // we will need to specify or "recommend" some standard bumps-per-inch
    // and first look for the signatures in that region
    int total_blocks, // how hard do we look
    int *present,
    float *scale,
    float *rotation,
    float *mellin_mag_transform
){
    int xblocks,yblocks,i,j,xlength,ylength;
    unsigned char *pdata;

    // the checking takes the first N 128by128 bump regions, FFT's them,
    // converts them to magnitudes, adds them all, then does
    // the fourier \mellin check between the added versions and
    // the master public grid FM profile.
    // A Year/No is generated based on the S/N found between a peak and the
    // background

    // find and use full integral blocks only, unless the data is shorter
    // than a full integral block
    int xumpszie = xdim/probable_bump_size;
    int yumpszie = ydim/probable_bump_size;
    xblocks = xumpszie / SIGNATURE_BLOCK_DIMENSION;
    yblocks = yumpszie / SIGNATURE_BLOCK_DIMENSION;

    // temporary
    total_blocks = xblocks * yblocks; // again, 0 will function

    // create the basic fourier magnitude array (SIGDIM*(SIGDIM/2+1)) or 128 by 65
    int n=SIGNATURE_BLOCK_DIMENSION;
    float *fourier_mag = new float[n*(1+n/2)]; // only stores the magnitude
    float *buffer = new float[n*(n+2)]; // give it a full array for processing inside 'add_block'
    int m = MELLIN_DIMENSION;

```

```

float *mellin_mag = new float[m*(m+2)];
float f0 = (float)0.0;
for(i=0;i<(n*(1+n/2));i++){fourier_mag[i]=f0;

int count = 0;
int truncated;
for(i=0;i<yblocks;i++){
    count++;
    for(j=0;j<xblocks;j++){
        pdata = &data[(i*xdim+j)*n*probable_bump_size]; // offset to this block
        if(xblocks == 0 || yblocks == 0){
            truncated = 1;
            if(xblocks==0)xlength = xumpszie;
            else xlength = n;
            if(yblocks==0)ylength = yumpszie;
            else ylength = n;
        }
        else {
            truncated = 0;
            xlength = n;
            ylength = n;
        }
        add_block_magnitude(
            pdata,
            fourier_mag,
            n,
            buffer,
            xlength,
            ylength,
            probable_bump_size,
            xdim, // pixel based jump pointer for moving down rows
            truncated
        );
        if(count >= total_blocks){j-xblocks;i-yblocks;}//this kicks it out
    }
}

// temporary: ship this one back for display
// use atemp.bmp as input alignment template file
//memcpy(mellin_mag_transform,fourier_mag,sizeof(float)*n*(n/2+1));
//return(i);

// now fourier mellinize the magnitude profile
log_polar_remap_public(fourier_mag,mellin_mag,n);
// temporary display results code
// use atemp128.bmp as input alignment template file
//memcpy(mellin_mag_transform,mellin_mag,sizeof(float)*n*n);
//return(i);

// fourier transform the dog
realfft2d_in_place(mellin_mag,7,0,wr,w1);

load_grid_family(i); // will immediately return if already done
// temporary display results code: this one has a corresponding return inside
load_grid_family
//memcpy(mellin_mag_transform,subliminal_grid,sizeof(float)*128*128);
//return(i);

// now compare the patterns
int bits = (int)(log((double)(n+1)) / log(2.0)); // fftdim should always be power
of 2

int number_candidates = 20;
float *rotation_buf = new float[number_candidates];
float *scale_buf = new float[number_candidates];
float *value = new float[number_candidates];

gmf(mellin_mag,mellin_mag_transform,n,bits,number_candidates,rotation_buf,scale_buf,value,0);
//return(i);

// a first crack at deciding whether or not a signature/grid is present is possible
// at this point: the ratio between value and value[n] should be above some
// threshold. If this is unreliable, then complete the alignment/read process.
// read the control bits and their checksums, and see if the checksums are right;
// this will obviously take a longer time to make a negative decision.

delete [] fourier_mag;
delete [] buffer;
delete [] mellin_mag;

float detection_value = value[0] / value[19];
float threshold_detect = (float)2.0; // where's our empirical data anyway, false-positive
curves, true double entendre negatives, etc.
if(detection_value > threshold_detect){ // we have a winna
    // if the suspect image has been rotated clockwise, rotation_buf will be positive
    // if the suspect image has been expanded, scale will come back negative
    rotation_buf[0] = (float)(90.0 / 128.0);
    double increment = pow(2.0, 0.025);

```



```

scale_buf[0] = (float)pow(increment, (double)scale_buf[0]);
if (xblocks == 0 || yblocks == 0) {
    truncated = 1;
    if (xblocks == 0) xlength = xbumpsize;
    else xlength = n;
    if (yblocks == 0) ylength = ybumpsize;
    else ylength = n;
}

// resolve 90 degree ambiguity in rotation/orientation
resolve_orientation(data, xlength, ylength, xdim, probable_bump_size,
    n_xdim, rotation_buf[0], &scale_buf[0]);

*rotation = rotation_buf[0];
*scale = scale_buf[0];
*present = 1;

// now find precise global alignment parameters
}
else { // send back no go on first detect, then get options for quitting or looking harder
    *present = 0;
}

delete () rotation_buf;
delete () scale_buf;
delete () value;

return (1);
}

int experiment(
    unsigned char *data,
    int n
) {
    float *imag = new float[n*n];
    // for (i=0; i<n*n; i++) imag[i] = (float)0.0;
    load_grid_family(); // will immediately return if already done
    realfft2d_in_place(subliminal_grid, 7.0, wr, w1);
    fft2d(subliminal_grid, imag, 7.0, wr, w1);
    return (1);
}

/* main registration program: to be used as main module inside other programs */
int align_direct_registration (
    unsigned char *template,
    int template_xdim,
    int template_ydim,
    unsigned char *suspect,
    int suspect_xdim,
    int suspect_ydim,
    int num_channels
) {
    if (1) {
        // experiment(ttemplate, template_xdim);
        // return (1);

        int present;
        float rotation, scale;
        extern float *mellin_mag_transform;
        hunt_for_grid(
            suspect,
            suspect_xdim,
            suspect_ydim,
            num_channels,
            1,
            10,
            &present,
            &scale,
            &rotation,
            mellin_mag_transform
        );

        // temporary: place mellin_mag_transform into ttemplate for return

```

```

// use atemp.bmp as input alignment template file
/*
float highest=(float)-1e20, lowest=(float)1e20;
int i,n=128;
for (i=0; i<(n*(n/2+1)); i++) {
    if (i/128 < 6) && (abs((i*128)-64) < 6) ;
    else {
        if (mellin_mag_transform[i]>highest) highest=mellin_mag_transform[i];
        if (mellin_mag_transform[i]<lowest) lowest=mellin_mag_transform[i];
    }
}
highest = (float)255.0/(highest-lowest);
for (i=0; i<(n*(n/2+1)); i++) {
    if (i/128 < 6) && (abs((i*128)-64) < 6) { ttemplate[i] = (unsigned char)100;
    else ttemplate[i] = (unsigned char) ( (mellin_mag_transform[i] - lowest ) * highest);
    }
}

// use atempl128.bmp as input alignment template file
float highest=(float)-1e20, lowest=(float)1e20;
int i,n=128;
for (i=0; i<(n*n); i++) {
    if (mellin_mag_transform[i]>highest) highest=mellin_mag_transform[i];
    if (mellin_mag_transform[i]<lowest) lowest=mellin_mag_transform[i];
}
highest = (float)255.0/(highest-lowest);
for (i=0; i<(n*n); i++) {
    ttemplate[i] = (unsigned char) ( (mellin_mag_transform[i] - lowest ) * highest);
}
}
else {

int i, fftdim, bits, array_size, lp_array_size;
int alignment_mode=2, downsample;
int number_candidates = MAX_CANDIDATES; /* number of peaks looked at */
float rotation(MAX_CANDIDATES), scale(MAX_CANDIDATES), value(MAX_CANDIDATES);
float x_trans(MAX_CANDIDATES), y_trans(MAX_CANDIDATES), x[5], y[5];
unsigned char *suspect_lum = new unsigned char[suspect_xdim*suspect_ydim];
unsigned char *template_lum = new unsigned char[template_xdim*template_ydim];

// if color image, then create collapse template into a single image.
// while the real suspect is used during final resampling
if (num_channels == 3) {
    unsigned char *pin,*ptemplate;
    ptemplate = template_lum;
    pin = ttemplate+1;
    for (i=0; i<(template_xdim*template_ydim); i++) {
        *ptemplate++ = *pin; // no need for extreme accuracy
        pin++;
    }
    ptemplate = suspect_lum;
    pin = suspect+1;
    for (i=0; i<(suspect_xdim*suspect_ydim); i++) {
        *ptemplate++ = *pin; // no need for extreme accuracy
        pin++;
    }

    // find working array size after downsampling (if downsampling is called at all)
    fftdim = get_working_dimension(alignment_mode, template_xdim, template_ydim,
        suspect_xdim, suspect_ydim, downsample);
    array_size = fftdim*fftdim*2;
    lp_array_size = lp_sampling*(lp_sampling*2); // the extra 2 is due to the fft routine
    being used
    bits = (int) (log( (double)(fftdim*1) ) / log( 2.0 )); // fftdim should always be power
    of 2

    // create the requisite arrays
    float *template_real = new float[array_size];
    float *template_lp_real = new float[lp_array_size];
    float *suspect_real = new float[array_size];
    float *suspect_lp_real = new float[lp_array_size];
    float *ttemp = new float[lp_array_size];
    float *suspect_copy = new float[array_size];

    // copy the two inputs into the arrays, with any downsampling and windowing applied
    if (num_channels == 1) {
        copy_downsample_window(suspect, suspect_xdim, suspect_ydim, suspect_real,
            fftdim, downsample);
        copy_downsample_window(ttemplate, template_xdim, template_ydim, template_real,
            fftdim, downsample);
    }
    else if (num_channels == 3) {
        copy_downsample_window(suspect_lum, suspect_xdim, suspect_ydim, suspect_real,
            fftdim, downsample);
        copy_downsample_window(template_lum, template_xdim, template_ydim, template_real,
            fftdim, downsample);
    }
}
else if (num_channels == 3) {
    copy_downsample_window(suspect_lum, suspect_xdim, suspect_ydim, suspect_real,
        fftdim, downsample);
    copy_downsample_window(template_lum, template_xdim, template_ydim, template_real,
        fftdim, downsample);
}
}

```

```

fftdim, downsamples);
}
memcpy(suspect_copy, suspect_real, array_size*sizeof(float) );
/* real-valued 2D FFT both suspect and template into it's half-planes complex self */
realfft2d_in_place(template_real, bits, 0, wr, wi);
realfft2d_in_place(suspect_real, bits, 0, wr, wi);
// calculate fourier mellin transform
fourier_mellin_transform(template_real, ftemp, fftdim, template_lp_real);
fourier_mellin_transform(suspect_real, ftemp, fftdim, suspect_lp_real);
/* assuming the inputs are both real only, then real 2D FFT each */
realfft2d_in_place(template_lp_real, lp_bits, 0, wr, wi);
realfft2d_in_place(suspect_lp_real, lp_bits, 0, wr, wi);
/* perform generalized matched filter on the two resulting arrays, outputting some number of
likely candidates, with their associated parameters */
gmf(template_lp_real, suspect_lp_real, lp_sampling, lp_bits, number_candidates,
rotation, scale, value, 0);
// change units on rotation and scale for later stages
for(i=0; i<number_candidates; i++){
rotation[i] *= ((float)180.0 / (float)lp_sampling); // converts to degrees
scale[i] = (float)pow((double)scale_increment, (double)scale[i]); // converts to linear scale
}
/* now we have a series of candidates ( or 1, and we just need to get the rotation
and translation information ) wherein one of them should be
the correct one; this next routine sifts through all candidates, including both
the nominal rotation scale and the state 180 degrees rotated from the nominal, and
finds which rotation, scale, and translation gives the highest matched filter
output; which then will be passed to the last fine tuning stage */
/* returns best candidate in first element of rotation, scale, x_trans, y_trans
get_best_candidate(number_candidates, ftemp, fftdim, bits, suspect_copy,
x_trans, y_trans, x_rot, y_rot, x_trans, y_trans, x_rot, y_rot, x_trans, y_rot, x_trans, y_rot);
suspect_ydim, downsamples, rotation, scale, x_trans, y_trans, template_real);
/* convert the scale/rotation/translation parameters of the downsampled arrays
into the x and y positions of the four corners of the suspect array, as projected
onto the template array. Precision in keeping track of the various coordinate systems
translated into final alignments to well better than a single pixel, especially
in light of the subtleties involved with downsampling. The four corners
are labeled 0 through 3 in the arrays x and y, where element 0 is the upper left corner
of the suspect, element 1 is the upper right, element 2 lower left, element 3 lower right.
The master 0 origin is placed at the upper left of the template array, while
the centerpoints of the two arrays play a role in rotations. The fifth
point in the array is the centerpoint, used just so you don't have to
recalculate it all the time */
get_corners_and_center(x, y, rotation[0], scale[0], x_trans[0], y_trans[0],
suspect_xdim, suspect_ydim, fftdim, downsamples);
/* now fine tune the result using tricky tricks, see notebook of Nov 28, 1995 */
if (num_channels == 1){
for(i=0; i<100; i++){
fine_tune_x_y(template_lum, template_xdim, template_ydim, suspect_xdim,
suspect_ydim, x, y, rotation);
}
}
else if (num_channels == 3){
fine_tune_x_y(template_lum, template_xdim, template_ydim, suspect_lum, suspect_xdim,
suspect_ydim, x, y, rotation);
}
}
/* last but not least, create the output image array, with various options */
final_image(template, template_xdim, template_ydim, suspect_xdim, suspect_ydim,
suspect_ydim, x, y, num_channels, 1); // '1' stands for aligned suspect with black everywhere else
/* Record some results of the alignment process in our status structure */
m_alignstatus.rotation = rotation[0];
m_alignstatus.x_scale = scale[0];
m_alignstatus.y_scale = scale[0];
m_alignstatus.x_trans = x_trans[0];
m_alignstatus.y_trans = y_trans[0];
/* free em all */
delete [] template_real;
delete [] template_lp_real;
delete [] suspect_real;
delete [] suspect_lp_real;
delete [] ftemp;
delete [] suspect_copy;
delete [] suspect_lum;
delete [] template_lum;
}
return(1);
}

```

```

/* shell to at least get the main registration program up and running, tested */
#define NEED_MAIN
////////////////////////////////////
//
// For Geoff's testing purposes, this main() function was used to
// create a stand-alone program, which exercised the alignment
// algorithms. This is "fixed", out for the windows version.
//
main( int argc, char *argv[] )
{
    int template_xdim,template_ydim,suspect_xdim,suspect_ydim,
    char template_filename[80],suspect_filename[80];
    FILE *inf;

    printf("\nTemplate file name please: ");
    scanf("%s",&template_filename);
    printf("\nx dimension and y dimension of template file: ");
    scanf("%d %d",&template_xdim,&template_ydim);
    printf("\nsuspect file name please: ");
    scanf("%s",&suspect_filename);
    printf("\nx dimension and y dimension of suspect file: ");
    scanf("%d %d",&suspect_xdim,&suspect_ydim);

    unsigned char *img = new unsigned char[template_xdim*template_ydim*sizeof(unsigned char)];
    unsigned char *img1 = new unsigned char[suspect_xdim*suspect_ydim*sizeof(unsigned char)];

    /* read in binary data into template */
    if (!inf) {
        printf(stderr,"register: can't open %s\n",template_filename);
        exit(1);
    }
    fread(img,sizeof(unsigned char),template_xdim*template_ydim,inf);
    fclose(inf);

    inf = fopen(suspect_filename,"rb");
    if (!inf) {
        printf(stderr,"register: can't open %s\n",suspect_filename);
        exit(1);
    }
    fread(img1,sizeof(unsigned char),suspect_xdim*suspect_ydim,inf);
    fclose(inf);

    /* returns registered image inside array 'template' */
    direct_registration(img,template_xdim,template_ydim,img1,suspect_xdim,suspect_ydim);

    /* write out binary data from template */
    inf = fopen("reg_out","wb");
    if (!inf) {
        printf(stderr,"register: can't open %s\n","reg_out");
        exit(1);
    }
    fwrite(img,sizeof(unsigned char),template_xdim*template_ydim,inf);
    fclose(inf);

    /* free and clean up */
    delete [] img;
    delete [] img1;

    return(0);
}

#endif //NEED_MAIN

```

[illegible]

```

#define ALIGN_H
// A structure used to define results of the alignment process.
typedef struct
{
    float rotation;
    float x_scale;
    float y_scale;
    float x_trans;
    float y_trans;
    float refinement;
} AlignStatus;

// Function prototypes: entry functions
class Align
{
public:
    Align();
    int direct_registration(unsigned char *template,
        int template_xdim,
        int template_ydim,
        unsigned char *suspect,
        int suspect_xdim,
        int suspect_ydim,
        int num_channels);
    // Accessor for status
    const AlignStatus GetAlignStatus(void) const {return m_alignStatus;}

private:
    // Private structure which contains results of alignment
    AlignStatus m_alignStatus;

    int fine_tune_x_y(unsigned char *template,
        int template_xdim,
        int template_ydim,
        unsigned char *suspect,
        int suspect_xdim,
        int suspect_ydim,
        float *x,
        float *y,
        float *rotation);
};

// Function prototypes: private functions
int gmf_identificat *real1,
float *real2,
float *imaginary1,
float *imaginary2,
int dim,
int bits,
float *offset);

#endif // ALIGN_H

// AlignDlg.cpp : implementation file
//
#include "stdafx.h"
#include "aligner.h"
#include "AlignDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// AlignDlg
IMPLEMENT_DYNAMIC(CAlignDlg, CFileDialog)

AlignDlg::AlignDlg(BOOL bOpenFileDialog, LPCTSTR lpszDefExt, LPCTSTR lpszFileName,
    DWORD dwFlags, LPCTSTR lpszFilter, CWnd* pParentWnd) :
    CFileDialog(bOpenFileDialog, lpszDefExt, lpszFileName, dwFlags, lpszFilter, pParentWnd)
{
}

BEGIN_MESSAGE_MAP(CAlignDlg, CFileDialog)
    //((AFX_MSG_MAP(CAlignDlg)
    // NOTE - the ClassWizard will add and remove mapping macros here.

```

```

// Seed the random number generator
srand(user_key);

// Image may be top to bottom or bottom to top.
// We must generate snow accordingly
if (bmHeader->biHeight > 0)
{
    bottom_up = TRUE;
    line = bmHeader->biHeight - 1;
}
else
{
    bottom_up = FALSE;
    line = 0;
}

// Generate snow one image scan line at a time.
for (line_cnt = 0; line_cnt < bmHeader->biHeight; line_cnt++)
{
    // Set pointer to first byte for this scan line.
    p_line = &image_data[line * (long) width_in_bytes];
    for (i = 0, j = 0; i < bmHeader->biWidth; i++)
    {
        if (bmHeader->biBitCount == 24)
        {
            // For 24 bit color case, need r,g,b snow...
            p_line[j++] = (char) rand();
            p_line[j++] = (char) rand();
            p_line[j++] = (char) rand();
        }
        else
        {
            // For test to make grey-scale and color keys match
            // we must call rand 3 times, but only keep same value
            // as the green channel of the rgb version. This way,
            // if we convert color image to greyscale we can read it.
            p_line[i] = (char) rand(); // we make grey snow same as green.
            rand();
        }
    }
    if (bottom_up) line--;
    else line++;
}

void CxKey::UseNewKey(unsigned newkey)
{
    char *line;
    int width_in_bytes, line_cnt, i;

    // Save the new key.
    user_key = newkey;

    width_in_bytes = (int) WIDTHBYTES(bmHeader->biWidth * bmHeader->biBitCount);

    // Seed the random number generator
    srand(user_key);

    for (line_cnt = 0; line_cnt < bmHeader->biHeight; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        line = &image_data[line_cnt * (long) width_in_bytes];
        for (i = 0; i < bmHeader->biWidth; i++)
        {
            line[i] = (char) rand();
        }
    }
}

//*****
// FILE: CxKey.h
// DESCRIPTION:
// The CxKey (for Coextensive Key) class encapsulates the functions and
// data structures used to generate a "snowy image" of the same extent
// (i.e., x, y dimensions) as the input image.
// This header file should be included by any module which creates or
// makes use of CxKey objects.
// CREATION DATE: August 15, 1995
// Copyright (c) 1995 Digimarc Incorporated, all rights reserved.
//*****

#endif COXKEY_H
#define COXKEY_H

#include "digimarc.h"
#include "Params.h"
#include "RavImage.h"
#include "stdafx.h"
#include "afx.h"

class CxKey
{
public:
    // The constructor is passed the user key value and ptrs to the DIB header
    // structures and the data space. The header is assumed to be filled out
    // correctly, while the data space is allocated but empty.
    // Alternative: pass an HDIB handle, allowing this class to handle locking.
    // FOR NOW, I ALSO ASSUME THE PALETTE HAS BEEN SET UP (its the same as image we are
    // signing)
    CxKey(int user_key, HDIB hDib);
    CxKey(unsigned user_key, BITMAPINFO *bmi, LPSTR lpDIBbits);

private:
    // Private member functions
private:
    // This function may be a useful idea for future, but it needs rework.
    // I'm making it private to assure no one is calling it.
    void UseNewKey(unsigned newkey);

private:
    // Private data
private:
    // Copy of the user key value.
    unsigned user_key;

    // Pointers to the bitmap info header structure, and the palette array.
    BITMAPINFOHEADER *bmiHeader; // pts to header structure
    RGBQUAD *bmiColors; // pts to beginning of palette array

    LPSTR lpDIBbits; // Pointer to DIB bits
    char *image_data; // Pointer to raw image data.
};

#endif COXKEY_H

//*****
// dibapi.cpp
// Source file for Device-Independent Bitmap (DIB) API. Provides
// the following functions:
// PaintDIB() - Painting routine for a DIB
// CreateDIBPalette() - Creates a palette from a DIB
// FindDIBbits() - Returns a pointer to the DIB bits
// DIBWidth() - Gets the width of the DIB
// DIBHeight() - Gets the height of the DIB
// PaletteSize() - Gets the size required to store the DIB's palette
// DIBNumColors() - Calculates the number of colors
// CopyHandle() - Makes a copy of the given global memory block
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or Winhelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.
//*****
#include "stdafx.h"
#include "dibapi.h"
#include <iostream>
#include <errno.h>
/*****
* PaintDIB()
* Parameters:
* *****/

```

```

    ::GlobalUnlock((HGLOBAL) hDIB);
    /* Reselect old palette */
    if (holdPal != NULL)
    {
        ::SelectPalette(hDC, holdPal, TRUE);
    }
    return bSuccess;
}

/*****
 * CreatedDIBPalette()
 * Parameter:
 *   hDIB hDIB - specifies the DIB
 * Return Value:
 *   HPALETTE - specifies the palette
 * Description:
 *   This function creates a palette from a DIB by allocating memory for the
 *   logical palette, reading and storing the colors from the DIB's color table
 *   into the logical palette, creating a palette from this logical palette,
 *   and then returning the palette's handle. This allows the DIB to be
 *   displayed using the best possible colors (important for DIBs with 256 or
 *   more colors).
 *****/

BOOL WINAPI CreatedDIBPalette(HDIB hDIB, CPalette* pPal)
{
    LPLOGPALETTE lpPal; // pointer to a logical palette
    HANDLE hLogPal; // handle to a logical palette
    HPALETTE hPal = NULL; // handle to a palette
    int i; // loop index
    WORD wNumColors; // number of colors in color table
    LPSTR lpbi; // pointer to packed-DIB
    LPBITMAPINFO lpbmi; // pointer to BITMAPINFO structure (Win3.0)
    LPBITMAPCOREINFO lpbmc; // pointer to BITMAPCOREINFO structure (old)
    BOOL bWinStyledDIB; // flag which signifies whether this is a Win3.0 DIB
    BOOL bResult = FALSE;

    /* if handle to DIB is invalid, return FALSE */
    if (hDIB == NULL)
        return FALSE;

    lpbi = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

    /* get pointer to BITMAPINFO (Win 3.0) */
    lpbmi = (LPBITMAPINFO) lpbi;

    /* get pointer to BITMAPCOREINFO (old 1.x) */
    lpbmc = (LPBITMAPCOREINFO) lpbi;

    /* get the number of colors in the DIB */
    wNumColors = ::DIBNumColors(lpbi);

    if (wNumColors != 0)
    {
        /* allocate memory block for logical palette */
        hLogPal = ::GlobalAlloc(GHND, sizeof(LOGPALETTE) * sizeof(PALENTRY) * wNumColors);

        /* if not enough memory, clean up and return NULL */
        if (hLogPal == 0)
        {
            ::GlobalUnlock((HGLOBAL) hDIB);
            return FALSE;
        }

        lpPal = (LPLOGPALETTE) ::GlobalLock((HGLOBAL) hLogPal);

        /* set version and number of palette entries */
        lpPal->palVersion = PALVERSION;
        lpPal->palNumEntries = (WORD) wNumColors;

        /* is this a Win 3.0 DIB? */
        bWinStyledDIB = IS_WIN30_DIB(lpbi);
        for (i = 0; i < (int) wNumColors; i++)
        {
            if (bWinStyledDIB)
            {

```

```

        lpPal->palPalEntry[i].peRed = lpBmi->bmiColors[i].rgbRed;
        lpPal->palPalEntry[i].peGreen = lpBmi->bmiColors[i].rgbGreen;
        lpPal->palPalEntry[i].peBlue = lpBmi->bmiColors[i].rgbBlue;
        lpPal->palPalEntry[i].peFlags = 0;
    }
    else
    {
        lpPal->palPalEntry[i].peRed = lpBmc->bmcColors[i].rgbRed;
        lpPal->palPalEntry[i].peGreen = lpBmc->bmcColors[i].rgbGreen;
        lpPal->palPalEntry[i].peBlue = lpBmc->bmcColors[i].rgbBlue;
        lpPal->palPalEntry[i].peFlags = 0;
    }
}

/* create the palette and get handle to it */
BRESULT = pPal->CreatePalette(lpPal);
::GlobalUnlock((HGLOBAL) hLogPal);
::GlobalFree((HGLOBAL) hLogPal);
}

::GlobalUnlock((HGLOBAL) hDIB);
return BRESULT;
}

/*.....*/
FINDDIBBits()
{
    Parameter:
    LPSTR lpbi - pointer to packed-DIB memory block
    Return Value:
    LPSTR - pointer to the DIB bits
    Description:
    This function calculates the address of the DIB's bits and returns a
    pointer to the DIB bits.
    /*.....*/
    LPSTR WINAPI FindDIBBits(LPSTR lpbi)
    {
        return (lpbi + ((LPDWORD)lpbi + ::PaletteSize(lpbi)));
    }
    /*.....*/
    Parameter:
    LPSTR lpbi - pointer to packed-DIB memory block
    Return Value:
    DWORD - width of the DIB
    Description:
    This function gets the width of the DIB from the BITMAPINFOHEADER
    which field if it is a Windows 3.0-style DIB or from the BITMAPCOREHEADER
    which field if it is an other-style DIB.
    /*.....*/
    WORD WINAPI DIBWidth(LPSTR lpDIB)
    {
        LPBITMAPINFOHEADER lpbmi; // pointer to a Win 3.0-style DIB
        LPBITMAPCOREHEADER lpbmc; // pointer to an other-style DIB
        /* point to the header (whether Win 3.0 and old) */
        lpbmi = (LPBITMAPINFOHEADER)lpDIB;
        lpbmc = (LPBITMAPCOREHEADER)lpDIB;
        /* return the DIB width if it is a Win 3.0 DIB */
        if (IS_WIN30_DIB(lpDIB))
            return lpbmi->biWidth;
        else /* it is an other-style DIB, so return its width */
            return (DWORD)lpbmc->bcWidth;
    }
}

/*.....*/
DIBHeight()
{
    Parameter:
    LPSTR lpbi - pointer to packed-DIB memory block
    Return Value:
    DWORD - height of the DIB
    Description:
    This function gets the height of the DIB from the BITMAPINFOHEADER
    height field if it is a Windows 3.0-style DIB or from the BITMAPCOREHEADER
    height field if it is an other-style DIB.
    /*.....*/
    DWORD WINAPI DIBHeight(LPSTR lpDIB)
    {
        LPBITMAPINFOHEADER lpbmi; // pointer to a Win 3.0-style DIB
        LPBITMAPCOREHEADER lpbmc; // pointer to an other-style DIB
        /* point to the header (whether old or Win 3.0) */
        lpbmi = (LPBITMAPINFOHEADER)lpDIB;
        lpbmc = (LPBITMAPCOREHEADER)lpDIB;
        /* return the DIB height if it is a Win 3.0 DIB */
        if (IS_WIN30_DIB(lpDIB))
            return lpbmi->biHeight;
        else /* it is an other-style DIB, so return its height */
            return (DWORD)lpbmc->bcHeight;
    }
}
/*.....*/
PaletteSize()
{
    Parameter:
    LPSTR lpbi - pointer to packed-DIB memory block
    Return Value:
    WORD - size of the color palette of the DIB
    Description:
    This function gets the size required to store the DIB's palette by
    multiplying the number of colors by the size of an RGBQUAD (for a
    Windows 3.0-style DIB) or by the size of an RGBTRIPLE (for an other-
    style DIB).
    /*.....*/
    WORD WINAPI PaletteSize(LPSTR lpbi)
    {
        /* calculate the size required by the palette */
        if (IS_WIN30_DIB(lpbi))
            return (WORD)((DIBNumColors(lpbi) * sizeof(RGBQUAD)));
        else
            return (WORD)((DIBNumColors(lpbi) * sizeof(RGBTRIPLE)));
    }
}
/*.....*/
DIBNumColors()
{
    Parameter:
    LPSTR lpbi - pointer to packed-DIB memory block
    Return Value:
    WORD - number of colors in the color table
    Description:
    This function calculates the number of colors in the DIB's color table
    by finding the bits per pixel for the DIB (whether Win3.0 or other-style
    DIB). If bits per pixel is 1: colors=2, if 4: colors=16, if 8: colors=256,
    if 24, no colors in color table.
    /*.....*/
}

```

```

.....
WORD WINAPI DIBNumColors(LPSTR lpbi)
{
    WORD wBitCount; // DIB bit count

    /* If this is a Windows-style DIB, the number of colors in the
       color table can be less than the number of bits per pixel
       allows for (i.e. lpbi->wBitCount can be set to some value).
       If this is the case, return the appropriate value.
    */

    if (IS_WIN30_DIB(lpbi))
    {
        DWORD dwClrUsed;

        dwClrUsed = ((LPBITMAPINFOHEADER)lpbi)->wBitCount;
        if (dwClrUsed != 0)
            return (WORD)dwClrUsed;
    }

    /* Calculate the number of colors in the color table based on
       the number of bits per pixel for the DIB.
    */
    if (IS_WIN30_DIB(lpbi))
        wBitCount = ((LPBITMAPINFOHEADER)lpbi)->wBitCount;
    else
        wBitCount = ((LPBITMAPCOREHEADER)lpbi)->wBitCount;

    /* return number of colors based on bits per pixel */
    switch (wBitCount)
    {
        case 1:
            return 2;
        case 4:
            return 16;
        case 8:
            return 256;
        default:
            return 0;
    }
}

.....
* DIBBitCount()
* Parameter:
* LPSTR lpbi - pointer to packed-DIB memory block
* Return Value:
* WORD - number of bits per pixel
* Description:
* Added by Clay Davidson 11/7/95. Simply returns the number of bits per
  pixel (i.e. 2, 4, 8, 24), regardless of the state of the color table.
  .....
WORD WINAPI DIBBitCount(LPSTR lpbi)
{
    WORD wBitCount;

    if (IS_WIN30_DIB(lpbi))
        wBitCount = ((LPBITMAPINFOHEADER)lpbi)->wBitCount;
    else
        wBitCount = ((LPBITMAPCOREHEADER)lpbi)->wBitCount;

    return wBitCount;
}

.....
// Clipboard support
// .....
// Function: CopyHandle (from SDK DIBview sample clipbrd.c)
// Purpose: Makes a copy of the given global memory block. Returns
//           a handle to the new memory block (NULL on error).
//           Routine stolen verbatim out of ShowDIB.
//

```

```

// Params: h -- Handle to global memory to duplicate.
// Returns: Handle to new global memory block.
// .....
HANDLE WINAPI CopyHandle (HANDLE h)
{
    BYTE *lpCopy;
    BYTE *lp;
    HANDLE hCopy;
    DWORD dwLen;

    if (h == NULL)
        return NULL;

    dwLen = ::GlobalSize((HGLOBAL) h);
    if ((hCopy = (HANDLE) ::GlobalAlloc (GMEM, dwLen)) != NULL)
    {
        lpCopy = (BYTE *) ::GlobalLock((HGLOBAL) hCopy);
        lp = (BYTE *) ::GlobalLock((HGLOBAL) h);
        while (dwLen--)
            *lpCopy++ = *lp++;
        ::GlobalUnlock((HGLOBAL) hCopy);
        ::GlobalUnlock((HGLOBAL) h);
    }
    return hCopy;
}

// dibapi.h
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#ifndef _INC_DIBAPI
#define _INC_DIBAPI

/* Handle to a DIB */
DECLARE_HANDLE(HDIB);

/* DIB constants */
#define PALVERSION 0x300

/* DIB Macros */

#define IS_WIN30_DIB(lpbi) ((LPDWORD)(lpbi)) == sizeof(BITMAPINFOHEADER)
#define RECTWIDTH(lprc) ((lprc)->right - (lprc)->left)
#define RECTHEIGHT(lprc) ((lprc)->bottom - (lprc)->top)

// WIDTHBYTES performs DWORD-aligning of DIB scanlines. The 'bits'
// parameter is the bit count for the scanline (biWidth * biBitCount),
// and this macro returns the number of DWORD-aligned bytes needed
// to hold those bits.

#define WIDTHBYTES(bits) (((bits) + 31) / 32 * 4)

/* Function prototypes */
BOOL WINAPI PaintDIB (HDC, LPRECT, HDIB, LPRECT, CPalette* pPal);
BOOL WINAPI CreateDIBPalette(HDIB, HDIB, CPalette* pPal);
LPSTR WINAPI FindDIBBits (LPSTR lpbi);
DWORD WINAPI DIBWidth (LPSTR lpbi);
WORD WINAPI DIBHeight (LPSTR lpbi);
WORD WINAPI DIBHeight (LPSTR lpbi);
WORD WINAPI DIBNumColors (LPSTR lpbi);
WORD WINAPI DIBBitCount (LPSTR lpbi);
HANDLE WINAPI CopyHandle (HANDLE h);

BOOL WINAPI SavedDIB (HDIB HDib, CFile* pFile);
HDIB WINAPI ReadDIBFile(CFile* pFile);

#endif // _INC_DIBAPI

```



```

static int irvb(int n, int b)
{
    register int r;
    register int i;
    register int nn;
    register int bb;

    bb = b;
    nn = n;

    switch( bb )
    {
        case 1 : return( t1(nn) );
        case 2 : return( t2(nn) );
        case 3 : return( t3(nn) );
        case 4 : return( t4(nn) );
        case 5 : return( t5(nn) );
        case 6 : return( t6(nn) );
        case 7 : return( t7(nn) );
        case 8 : return( t8(nn) );
        case 9 : return( t9(nn) );
        case 10 : return( t10(nn) );
        default:
            r = 0;
            for( i = 0; i < bb; i++ )
            {
                r = r << 1;
                r = r | ( nn & 1 );
                nn = nn >> 1;
            }
            return( r );
    }
}

/* fft() is a routine that calculates the discrete Fourier transform
* of two arrays taken to be the real and the imaginary parts of an
* complex array. It returns the transform in the arrays.
*/
void fft(float *ar, float *ai, int nbits, int inv, float *wr, float *wi, int neww)
{
    float *ar; /* the real part of the array */
    float *ai; /* the imag part of the array */
    int nbits; /* log base 2 of the number of elements in the arrays */
    int inv; /* nonzero to indicate the inverse transform */
    float *wr; /* the real part of an array of coefficients */
    float *wi; /* the imag part of an array of coefficients */
    int neww; /* nonzero to indicate the coefficients must be calcd */

    register float *aar;
    register float *aai;
    register float *pr1;
    register float *pi1;
    register float *pr2;
    register float *pi2;
    register float *r1;
    register float *i1;
    register float *r2;
    register float *i2;
    register float *wreal;
    register float *wimag;
    register float *pwr;
    register float *pwi;
    register float *w;

    aar = ar;
    aai = ai;

    n = 1 << nbits;
    fn = (float) n;

    if( inv == 0 )

```

```

    {
        for( i = 0; i < n; i++ )
        {
            aar[i] = aar[i] / fn;
            aai[i] = -aai[i] / fn;
        }
    }

    if( neww != 0 )
    {
        tpin = (float) 6.283186 / fn;
        n2 = n / 2;

        for( nb = 0; nb < n2; nb++ )
        {
            w = tpin * ( (float) irvb( nb, nbbits-1 ) );
            wr[nb] = (float) cos( double)w;
            wi[nb] = (float) sin( double)w;
        }

        nblock = 1;
        nsep = n;

        for( ns = 0; ns < nbits; ns++ )
        {
            nsep2 = nsep;
            nsep = nsep / 2;

            pwr = wr;
            pw1 = wi;

            for( nb=0; nb < nblock; nb++, pwr++, pw1++ )
            {
                n1 = nb*nsep2;
                n2 = n1+nsep;
                pr1 = aar[n1];
                pr2 = aar[n2];
                pi1 = aai[n1];
                pi2 = aai[n2];
                wreal = *pwr;
                wimag = *pwi;

                for( j=0; j<nsep; j++ )
                {
                    r1 = *pr1; r2 = *pr2; i1 = *pi1; i2 = *pi2;
                    areal = wreal * r2 - wimag * i2;
                    aimag = wimag * r2 + wreal * i2;
                    *(pr2++) = r1 - areal;
                    *(pi2++) = i1 - aimag;
                    *(pr1++) = r1 + areal;
                    *(pi1++) = i1 + aimag;
                }
            }

            nblock = nblock*2;
        }

        for( i = 0; i < n; i++ )
        {
            j = irvb( i, nbits );
            if( i < j )
            {
                areal = aar[i];
                aimag = aai[i];
                aar[i] = aar[j];
                aai[i] = aai[j];
                aar[j] = areal;
                aai[j] = aimag;
            }

            if( inv == 0 ) aai[i] = -aai[i];
        }

        int fft2d(float *ar, float *ai, int nbits, int inv, float *wr, float *wi)
        {
            int i;
            int j;
            int ii;
            int jj;
            int n;
            float *ar;
            float *ai;

            n = 1 << nbits;

            for( i = 1; i < n; i++ )
            {
                for( j = 0; j < i; j++ )

```

```

ij = (i<nbits)+j;
ji = (j<nbits)+i;
xr = ar[ij];
xi = ai[ij];
ar[ij] = ar[ji];
ai[ij] = ai[ji];
ar[ji] = xr;
ai[ji] = xi;
}

fft( sar[0], sai[0], nbits, inv, wr, wi, 1 );
for( i = 1 ; i < n ; i++ )
{
    fft( sar[i<nbits], sai[i<nbits], nbits, inv, wr, wi, 0 );
}

for( i = 1 ; i < n ; i++ )
{
    for( j = 0 ; j < i ; j++ )
    {
        ij = (i<nbits)+j;
        ji = (j<nbits)+i;
        xr = ar[ij];
        xi = ai[ij];
        ar[ij] = ar[ji];
        ai[ij] = ai[ji];
        ar[ji] = xr;
        ai[ji] = xi;
    }
}

for( i = 0 ; i < n ; i++ )
{
    fft( sar[i<nbits], sai[i<nbits], nbits, inv, wr, wi, 0 );
}

return(0);

```

void realfft_two_arrays(float *array1, float *array2, int nbits, int inv, float *wr, float *wi, int neww)

```

{
    register int j;
    register int n;
    register int nhalf;
    float temp1[MAX_LINEAR_DIMENSION], temp2[MAX_LINEAR_DIMENSION];
    register float *ptemp1;
    register float *ptemp2;
    register float *par;
    register float *pai;
    register float *pail;
    register float *ptemp1_i;
    register float *ptemp2_i;

    n = 1 << nbits;
    nhalf = n/2;
}

```

```

if(!inv){
    fft(array1,array2,nbits,inv,wr,wi,neww);
    /* sort the results */
    ptemp1 = temp1;
    ptemp2 = temp2;
    par = array1;
    pai = array2;
    *ptemp1 = *(par++);
    *ptemp2 = *(pai++);
    pail = &array1[n-i];
    ptemp1+=2;
    ptemp2+=2;
    for(j=1;j<nhalf;j++){
        *ptemp1++ = (float)0.5 * (*par + *pail);
        *ptemp2++ = (float)0.5 * (*pai + *pail);
        *ptemp1++ = (float)0.5 * (*pai - *pail);
        *ptemp2++ = (float)0.5 * (*par - *pail);
        par++,pail--;pai++,pail--;
    }
    temp1[i] = *par;
    temp2[i] = *pai;
    /* now copy the results back into original arrays */
    memcpy(array1,temp1,n*sizeof(float));
    memcpy(array2,temp2,n*sizeof(float));
}
else {
    /* re-sort results */
    ptemp1 = temp1;
    ptemp2 = temp2;
    par = array1;
}

```

```

pai = array2;
*ptemp1++ = *par;
*ptemp2++ = *pai;
par++;
pai++;
ptemp1_1 = stemp1[n-1];
ptemp2_1 = stemp2[n-1];
for(j=1;j<(n/2);j++){
    *ptemp1++ = (*par - *(pai+1)) ;
    *ptemp1_1-- = *(pai+1);
    *ptemp2++ = (*par+1) + *pai;
    *ptemp2_1-- = (-*(par+1) + *pai);
    par++;
    pai++;
}
*ptemp1 = array1[1];
*ptemp2 = array2[1];
fft(array1,array2,nbits,inv,wr,wi,neww);
}

```

/* this routine requires that the input array have two more rows of n appended, into which the nyquist row will be placed */

```

int realfft3_in_place(float *ar,int nbits,int inv,float *wr,float *wi)
{
    register int i;
    register int j;
    register int ij;
    register int ji;
    register int n;
    register int n2;
    register int nhalf;
    register float xr;
    register float xi;
    register float xri;
    register float xil;
    float temp_r[MAX_LINEAR_DIMENSION],temp_i[MAX_LINEAR_DIMENSION];
    register float *ptemp_r;
    register float *ptemp_i;
    register float *par;
    register float *pai;
    register float *pail;
    register float *ptemp_r1;
    register float *ptemp_i1;

    n = 1 << nbits;
    n2 = n*2;
    nhalf = n/2;

    if(!inv){
        /* pre-transpose */
        for( i = 1 ; i < n ; i++ )
        {
            for( j = 0 ; j < i ; j++ )
            {
                ij = (i<nbits)+j;
                ji = (j<nbits)+i;
                xr = ar[ij];
                ar[ij] = ar[ji];
                ar[ji] = xr;
            }
        }

        for( i = 0 ; i < nhalf ; i++ )
        {
            if(i==0)fft( sar[0], sai[n], nbits, inv, wr, wi, 1 );
            else fft( sar[n2-i], sai[n2-i+n], nbits, inv, wr, wi, 0 );

            /* sort and pack results */
            ptemp_r = temp_r;
            ptemp_i = stemp_i[2];
            par = &ar[n2-i];
            pail = &ar[n2-i+n];
            *ptemp_r++ = *(par++);
            *ptemp_r++ = *(par--);
            pai = &ar[n2-i+n];
            pail = &ar[n2-i+n2-1];
            for(j=i;j<nhalf;j++){
                *ptemp_r++ = (float)0.5 * (*par + *pail);
                *ptemp_i++ = (float)0.5 * (*pai + *pail);
                *ptemp_r++ = (float)0.5 * (*pai - *pail);
                *ptemp_i++ = (float)0.5 * (*par - *pail);
                par++,pail--;pai++,pail--;
            }
            temp_i[0] = *par;
        }
    }
}

```

```

temp_i[1] = *pai;

/* now copy the results back into original arrays */
memcpy(&ar[n2*i:n], temp_r, n*sizeof(float));
memcpy(&ar[n2*i:n], temp_i, n*sizeof(float));
}

/* transpose */
for( i = 2; i < n; i+=2 ) {
    for( j = 0; j < i; j+=2 ) {
        ij = (i<nbits)+j;
        ji = (j<nbits)+i;
        xr = ar[ij];
        xi = ar[ji+n];
        xri = ar[ij+1];
        xri = ar[ij+1+n];
        xli = ar[ji+1+n];
        ar[ij] = ar[ji];
        ar[ij+n] = ar[ji+n];
        ar[ij+1] = ar[ji+1];
        ar[ij+n+1] = ar[ji+n+1];
        ar[ji] = xr;
        ar[ji+n] = xi;
        ar[ji+1] = xri;
        ar[ji+n+1] = xli;
    }
}

/* place nyquist row into n'n row, and zero out their imaginary rows */
memcpy(&ar[n'n], &ar[n], n*sizeof(float));
memset(&ar[n], 0, n*sizeof(float));
memset(&ar[n'n+n], 0, n*sizeof(float));

for( i = 0; i < nhalf+1; i++ ) fft( &ar[n2*i], &ar[n2*i+n], nbits, inv, wr, wi, 0 );

/* finally, shift the arrays in order to simplify external processing */
for( i=0; i<n+2; i++ ) {
    memcpy( temp_r, &ar[i*n], nhalf*sizeof(float) );
    memcpy( &ar[i*n], &ar[nhalf*i*n], nhalf*sizeof(float) );
    memcpy( &ar[nhalf*i*n], temp_r, nhalf*sizeof(float) );
}

else {
    /* undo format */
    for( i=0; i<(n+2); i++ ) {
        memcpy( temp_r, &ar[i*n], (n/2)*sizeof(float) );
        memcpy( &ar[i*n], &ar[n/2+i*n], (n/2)*sizeof(float) );
        memcpy( &ar[n/2+i*n], temp_r, (n/2)*sizeof(float) );
    }

    fft( &ar[0], &ar[n], nbits, inv, wr, wi, 1 );
    for( i = 1; i < (1+n/2); i++ ) fft( &ar[(2-i)*n], &ar[(2-i)*n+n], nbits, inv, wr, wi, 0 );
    memcpy( &ar[n], &ar[n'n], n*sizeof(float) );

    /* transpose */
    for( i = 2; i < n; i+=2 ) {
        for( j = 0; j < i; j+=2 ) {
            ij = (i<nbits)+j;
            ji = (j<nbits)+i;
            xr = ar[ij];
            xi = ar[ji+n];
            xri = ar[ij+1];
            xri = ar[ij+1+n];
            xli = ar[ji+1+n];
            ar[ij] = ar[ji];
            ar[ij+n] = ar[ji+n];
            ar[ij+1] = ar[ji+1];
            ar[ij+n+1] = ar[ji+n+1];
            ar[ji] = xr;
            ar[ji+n] = xi;
            ar[ji+1] = xri;
            ar[ji+n+1] = xli;
        }
    }

    /* re-sort results */
    ptemp_r = temp_r;
    ptemp_i = temp_i;
    par = &ar[(2*i):n];
    *(&ptemp_r++) = *(&par++);
    *(&ptemp_i++) = *(&par++);

    pai = &ar[(2+i)*n];
    ptemp_r1 = &temp_r[n-1];
    ptemp_i1 = &temp_i[n-1];
    for( j=1; j<(n/2); j++ ) {
        *(&ptemp_r++) = *(&par + *(&pai+1) );
        *(&ptemp_i1++) = *(&par + *(&pai+1) );
    }
}

```

```

    m_hDIB = hDIB;
    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0];

    // Set the pointer to the image data.
    m_hpDIBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;

    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);

    // Image (hDIB hDIB)
    // Constructor which creates an image object, given the name of a DIB
    // or BMP file.
    Image::Image(CString filename)
    {
        CFile file;
        CFileException fe;
        BITMAPINFO *bmi_info;
        m_hPackedData = NULL;

        if (!file.Open(filename, CFile::modeRead | CFile::shareDenyWrite, &fe))
        {
            CString msg("Error reading image file: ");
            msg += filename;
            MessageBox(NULL, msg, NULL, MB_ICONINFORMATION | MB_OK);
            m_fileOK = FALSE;
        }
        else
            m_fileOK = TRUE;

        // Try to read the DIB file, catch any exceptions.
        TRY
        {
            m_hDIB = ::ReadDIBFile(file);
        }
        CATCH(CFileException, eLoad)
        {
            file.Abort();
            MessageBox(NULL, "Error reading the image file", NULL,
                MB_ICONINFORMATION | MB_OK);
            m_hDIB = NULL;
            m_fileOK = FALSE;
        }
        END_CATCH

        m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

        // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
        // WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
        // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

        bmi_info = (BITMAPINFO *) m_lpDIB;
        // Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
        m_lpBmiHeader = &bmi_info->bmiHeader;
        m_lpBmiColors = &bmi_info->bmiColors[0];

        // Set the pointer to the image data.
        m_hpDIBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

        m_BitsPerPixel = m_lpBmiHeader->biBitCount;
        m_XDim = m_lpBmiHeader->biWidth;
        m_YDim = m_lpBmiHeader->biHeight;
        m_Compression = m_lpBmiHeader->biCompression;
        m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);

        // --image()
    }
}

// The destructor for the Image class of objects.
// Image::~Image(void)
{
    ::GlobalUnlock( (HGLOBAL) m_hDIB);

    if (m_hPackedData != NULL)
    {
        ::GlobalUnlock( (HGLOBAL) m_hPackedData);
        ::GlobalFree( (HGLOBAL) m_hPackedData);
    }
}

// MakePackedData()
// This function copies the DIB image data into a packed format. This
// is important for two reasons: 1) the DIB formatted data is arranged
// so that each scan line starts on a long word boundary, so there may
// be up to 3 unused bytes at the end of each scan line in the case of
// 8 bit data. This arrangement is inconvenient when passing the image
// data to the core algorithms. Also, 2) if a palette is being used
// (this is the case for all but 24 bit image data), this routine looks
// up the actual image values using the palette and places these values
// in the packed data array. The member variable m_hPackedData is the
// handle to the packed data.
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
void Image::MakePackedData(void)
{
    unsigned char *hpline;
    unsigned char *hpData;
    int line_cnt, line, i;
    BOOLEAN bottom_up;

    // Create space and get handle for the packed data of the image.
    m_hPackedData = ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
        (m_XDim * (long) m_YDim));
    if (m_hPackedData == 0)
        AfxThrowMemoryException();

    // Lock the packed data global memory (leave locked until destructor).
    m_hPackedData = (unsigned char *) ::GlobalLock( (HGLOBAL) m_hPackedData);

    hpData = m_hPackedData;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff, don't let it correct for bottom_up
    bottom_up = FALSE;
    line = 0;

    // Now go through each line and create the packed array.
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpline = &m_hpDIBits[line * (long) m_WidthInBytes];
        for (i = 0; i < m_XDim; i++)
        {
            if (m_BitsPerPixel == 24)
                *hpData++ = hpline[i];
            else
            {
                // For 8 bit (and any other non 24 bit data) we
                // take the image data to be indices into the color
                // table. We look up the actual value. Note we
                // assume grey-scale (i.e., r,g,b triples are all equal -
                // we read the green.
                *hpData++ = m_lpBmiColors[hpline[i]].rgbGreen;
            }
        }
        if (bottom_up) line--;
        else line++;
    }
}

```

```

////////////////////
// UnpackData()
//
// This function moves the contents of the packed data array back into
// the DIB data space. This would be used, for example, after one the
// core algorithms have been used to update the data in the packed array,
// and we want to update the DIB to reflect the changes. Note that this
// requires that we create our own palette, since otherwise we don't know
// that the new data values have corresponding entries in the palette.
//
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
void Image::UnpackData(void)
{
    unsigned char *hpline;
    unsigned char *hpdata;
    int line_cnt, line, i;
    BOOLEAN bottom_up;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff: don't let it correct for bottom_up
    bottom_up = FALSE;
    line = 0;

    hpdata = m_hpPackedData;
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpline = m_hpDIBbitsline * (long) m_WidthInBytes;
        for (i = 0; i < m_XDim; i++)
        {
            hpline[i] = *hpdata++;
        }
        if (bottom_up) line--;
        else line++;
    }

    // Next, we force the palette to be our standard 8 bit grey-scale
    // palette.
    if (m_BitsPerPixel == 8)
    {
        // Set ptr to beginning of palette
        LRGBQUAD pal = m_lpBmiColors;

        for (i = 0; i < 256; i++)
        {
            pal[i].rgbBlue = pal[i].rgbGreen = pal[i].rgbRed = i;
        }

        else
        {
            MessageBox(NULL, "Can only unpack 8 bit image data", NULL,
                MB_ICONEXCLAMATION | MB_OK);
        }
    }
}

IMAGE.CPP
////////////////////
// File: Image.cpp
//
// Contains the implementation for the Image class. Image objects
// are used to contain the image data, and provide a more convenient
// set of services related to accessing the image data as well as
// attribute variables describing the image.
//
// Include "Image.h"
#include "dibapi.h"
#include "stdafx.h"

////////////////////
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given a handle to
// a DIB which is already in memory.

```

```

////////////////////
// Image(HDIB hDIB)
//
// BITMAPINFO *bmi_info;
//
// m_hpPackedData = NULL;
// m_fileOK = TRUE;
//
// m_hDIB = hDIB;
//
// m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);
//
// NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
// WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
// I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.
//
// Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
m_lpBmiHeader = &bmi_info->bmiHeader;
m_lpBmiColors = &bmi_info->bmiColors[0]; // will be null for 24 bit
//
// Set the pointer to the image data.
m_hpDIBbits = (unsigned char *) ::FindDIBbits(m_lpDIB);
//
// m_BitsPerPixel = m_lpBmiHeader->biBitCount;
// m_XDim = m_lpBmiHeader->biWidth;
// m_YDim = m_lpBmiHeader->biHeight;
// m_Compression = m_lpBmiHeader->biCompression;
//
// m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
//
//
//
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given the name of a DIB
// or BMP file.
//
// Image(CString filename)
//
// File
// filename;
// CFileException fe;
// BITMAPINFO *bmi_info;
//
// m_hpPackedData = NULL;
//
// if (!file.Open(filename, CFile::modeRead | CFile::shareDenyWrite, &fe))
// {
//     CString msg("Error reading image file: ");
//     msg += filename;
//     MessageBox(NULL, msg, NULL, MB_ICONINFORMATION | MB_OK);
//     m_fileOK = FALSE;
// }
// else
// {
//     m_fileOK = TRUE;
// }
//
// Try to read the DIB file, catch any exceptions.
//
// Try
// {
//     m_hDIB = ::ReadDIBfile(file);
// }
// CATCH(CFileException, eLoad)
// {
//     file.Abort();
//     MessageBox(NULL, "Error reading the image file", NULL,
//         MB_ICONINFORMATION | MB_OK);
//     m_hDIB = NULL;
//     m_fileOK = FALSE;
// }
// END_CATCH
//
// m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);
//
// NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
// WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
// I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.
//
// Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
bmi_info = (BITMAPINFO *) m_lpDIB;
// Set up a pointer to the BITMAPINFOHEADER;
m_lpBmiHeader = &bmi_info->bmiHeader;
m_lpBmiColors = &bmi_info->bmiColors[0];
//
// Set the pointer to the image data.
m_hpDIBbits = (unsigned char *) ::FindDIBbits(m_lpDIB);
//
// m_BitsPerPixel = m_lpBmiHeader->biBitCount;
// m_XDim = m_lpBmiHeader->biWidth;

```

```

    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;
    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

// -----
// -Image()
// -----
// The destructor for the Image class of objects.
Image::~Image(void)
{
    ::GlobalUnlock( (HGLOBAL) m_hBIB );
    if (m_hPackedData != NULL)
    {
        ::GlobalUnlock( (HGLOBAL) m_hPackedData );
        ::GlobalFree( (HGLOBAL) m_hPackedData );
    }
}

// -----
// MakePackedData()
// -----
// This function copies the DIB image data into a packed format. This
// is important for two reasons, 1) the DIB formatted data is arranged
// so that each scan line starts on a long word boundary, so there may
// be up to 3 unused bytes at the end of each scan line in the case of
// 8 bit data. This arrangement is inconvenient when passing the image
// data to the core algorithms. Also, 2) if a palette is being used
// (this is the case for all but 24 bit image data), this routine looks
// up the actual image values using the palette and places these values
// in the packed data array. The member variable m_hPackedData is the
// handle to the packed data.
//
// The force_to_1_chan argument is an optional boolean. It defaults
// to FALSE (see function prototype in image.h). If set to TRUE,
// only 1 channel of packed data is created, even if the image is 3
// channels. This is useful when creating snowy images from RGB
// images, since we currently always want 1 channel snowy images.
//
// void Image::MakePackedData(BOOLEAN force_to_1_chan)
//
// unsigned char *hpLine;
// unsigned char *hpData;
// int line_cnt, line, i, j;
// long size;
// BOOLEAN bottom_up;

// Create space and get handle for the packed data of the image.
size = m_YDim * m_XDim;
// For 24 bit true color, we will pack R,G,B values, so triple the size.
if (m_BitsPerPixel == 24 && force_to_1_chan == FALSE)
    size *= 3;
m_hPackedData = ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, size);
if (m_hPackedData == 0)
    AfxThrowMemoryException();

// Lock the packed data global memory (leave locked until destructor).
m_hPackedData = (unsigned char *)::GlobalLock( (HGLOBAL) m_hPackedData );
hpData = m_hPackedData;

// Image may be top to bottom or bottom to top.
if (m_lpBmiHeader->biHeight > 0)
{
    bottom_up = TRUE;
    line = m_YDim - 1;
}
else
{
    bottom_up = FALSE;
    line = 0;
}

// TEST CODE
// For Geoff, don't let it correct for bottom_up
// bottom_up = FALSE;
// line = 0;

// Now go through each line and create the packed array.
for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
{
    // Set pointer to first byte for this scan line.
    hpLine = &m_hPackedData[line * (long) m_WidthInBytes];
    for (i = 0; i < m_XDim; i++)
    {
        if (m_BitsPerPixel == 24)
        {
            hpLine[i*2] = *hpData++; // red
            hpLine[i*2+1] = *hpData++; // green
            hpLine[i*2+2] = *hpData++; // blue
            j += 3;
        }
        else
        {
            hpLine[i] = *hpData++;
        }
        if (bottom_up) line--;
        else line++;
    }
    // Next, we force the palette to be our standard 8 bit grey-scale
    // palette.
}

```



```

DECLARE_DYNAMIC(CMyChildWnd)
protected:
    CMyChildWnd(); // protected constructor used by dynamic creation
// Attributes
public:
// Operations
// Implementation
protected:
    virtual ~CMyChildWnd();

    virtual BOOL PreCreateWindow(CREATESTRUCT &cs);

// Generated message map functions
//{{AFX_MSG(CMyChildWnd)
// NOTE - the ClassWizard will add and remove member functions here.
// AFX_MSG
DECLARE_MESSAGE_MAP()
};

////////////////////////////////////

myfile.cpp

// Source file for Device-Independent Bitmap (DIB) API. Provides
// the following functions:
//
// SaveDIB() - Saves the specified dib in a file
// ReadDIBFile() - Loads a DIB from a file
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.
#include "stdafx.h"
#include <math.h>
#include <iostream>
#include <direct.h>
#include "dibapi.h"

/*
 * DIB Header Marker - used in writing DIBs to files
 */
#define DIB_HEADER_MARKER ((WORD) ('M' << 8) | 'B')

//.....
//
// SaveDIB()
//
// Saves the specified DIB into the specified CFile. The CFile
// is opened and closed by the caller.
//
// Parameters:
//
// hDIB hDib - Handle to the dib to save
//
// CFile* file - open CFile used to save DIB
//
// Return value: TRUE if successful, else FALSE or CFileException
//.....
//
// BOOL WINAPI SaveDIB(HDIB hDib, CFile* file)
//
// BITMAPFILEHEADER bmfHdr; // Header for Bitmap file
// LPBITMAPINFOHEADER lpBI; // Pointer to DIB info structure
// DWORD dwDIBSize;
//
// if (hDib == NULL)
//     return FALSE;
//
// * Get a pointer to the DIB memory, the first of which contains
// * a BITMAPINFO structure

```

```

//
lpBI = (LPBITMAPINFOHEADER) ::GlobalLock((HGLOBAL) hDib);
if (lpBI == NULL)
    return FALSE;

if (IIS_WIN30_DIB(lpBI))
{
    ::GlobalUnlock((HGLOBAL) hDib);
    return FALSE; // It's an other-style DIB (save not supported)
}

/*
 * Fill in the fields of the file header
 */

/* Fill in file type (first 2 bytes must be "BM" for a bitmap) */
bmfHdr.bfType = DIB_HEADER_MARKER; // "BM"

// Calculating the size of the DIB is a bit tricky (if we want to
// do it right). The easiest way to do this is to call GlobalSize()
// on our global handle, but since the size of our global memory may have
// been padded a few bytes, we may end up writing out a few too
// many bytes to the file (which may cause problems with some apps).
//
// So, instead let's calculate the size manually (if we can)
//
// First, find size of header plus size of color table. Since the
// first DWORD in both BITMAPINFOHEADER and BITMAPCOREHEADER contains
// the size of the structure, let's use this.
dwDIBSize = (LPDWORD)lpBI + ::Palettesize((LPSTR)lpBI); // Partial Calculation

// Now calculate the size of the image
if ((lpBI->biCompression == BI_RGB) || (lpBI->biCompression == BI_RGBA))
{
    // It's an RGB bitmap, we can't calculate size, so trust the
    // biSizeImage field
    dwDIBSize += lpBI->biSizeImage;
}
else
{
    DWORD dwBmBitsize; // Size of Bitmap Bits only
    // It's not RGB, so size is Width (DWORD aligned) * Height
    dwBmBitsize = WIDTHBYTES((lpBI->biWidth)*((DWORD)lpBI->biBitCount)) * lpBI->biHeight;
    dwDIBSize += dwBmBitsize;
}

// Now, since we have calculated the correct size, why don't we
// fill in the biSizeImage field (this will fix any .BMP files which
// have this field incorrect).
lpBI->biSizeImage = dwBmBitsize;

// Calculate the file size by adding the DIB size to sizeof(BITMAPFILEHEADER)
bmfHdr.bfSize = dwDIBSize + sizeof(BITMAPFILEHEADER);
bmfHdr.bfReserved1 = 0;
bmfHdr.bfReserved2 = 0;

/*
 * Now, calculate the offset the actual bitmap bits will be in
 * the file -- It's the Bitmap file header plus the DIB header,
 * plus the size of the color table.
 */
bmfHdr.bfOffBits = (DWORD)sizeof(BITMAPFILEHEADER) + lpBI->biSize
    + Palettesize((LPSTR)lpBI);

TRY
{
    // Write the file header
    file.Write((LPSTR)bmfHdr, sizeof(BITMAPFILEHEADER));
    // Write the DIB header and the bits
    //
    file.WriteHuge(lpBI, dwDIBSize);
}
CATCH (CFileException, e)
{
    ::GlobalUnlock((HGLOBAL) hDib);
    THROW_LAST();
}
END_CATCH

::GlobalUnlock((HGLOBAL) hDib);

```

```

    * Copyright (c) 1995 Digimarc Incorporated, all rights reserved. *
    *.....*/
    #include "stdafx.h"
    #include "packmsg.h"
    #include "string.h"
    #include "ctype.h"

    typedef char * Compact_Msg;

    //////////////////////////////////////
    // PackedMsg(const char *user_msg)
    //////////////////////////////////////
    // This is the PackedMsg constructor which is given an ASCII
    // message for use by the signer. It creates an array of
    // packed characters (a more compact representation than
    // ASCII), computes the checksum for the compact string,
    // and then creates a bit array containing the compact
    // message (this is the form the signer core algorithms
    // require).
    // PackedMsg::PackedMsg(const char *user_msg)
    {
        m_correctBits = 0;
        m_checksum = 0;
        m_recoveredChecksum = 0;
        m_computedReaderChecksum = 0;

        // Save the length, and a copy of the original user (ascii) message.
        m_msgLength = strlen(user_msg);
        m_asciiMsg = new char[m_msgLength+1];
        strcpy(m_asciiMsg, user_msg); // Note it is null terminated.
        m_recoveredAsciiMsg = new char[m_msgLength+1];

        // Allocate space for the packed message. Note there's no NULL termination.
        m_compactMsg = new char[m_msgLength];

        // Call the function which translates to compact form.
        PackMessage();

        // Compute the checksum of the compact message string
        m_checksum = ComputeChecksum(m_compactMsg, m_msgLength);

        // Allocate space for the MsgBitArray, which puts one bit of the
        // packed message in each char of an unsigned char array (this is
        // the format that the current core signer needs.
        // Also, we include space for checksum, same length as 1 char.
        // Also allocate space for the ReaderBitArray, which reader will use.
        m_msgBitArrayLength = (m_msgLength+1) * PACKED_BITS_PER_CHAR;
        m_msgBitArray = new unsigned char[m_msgBitArrayLength];
        m_readerBitArray = new unsigned char[m_msgBitArrayLength];

        unsigned char *p_bit_array = m_msgBitArray;
        unsigned char *p_reader_array = m_readerBitArray;
        int i, j;
        unsigned char mask;
        for (i = 0; i < m_msgLength; i++)
        {
            for (j = PACKED_BITS_PER_CHAR - 1; j >= 0; j--)
            {
                mask = 1 << j;
                if (m_compactMsg[i] & mask)
                    *p_bit_array = 1;
                else
                    *p_bit_array = 0;

                p_bit_array++;
                *p_reader_array++ = 0; // clear the readers array.
            }
        }

        // Continue by putting the checksum in the final PACKED_BITS_PER_CHAR
        // elements of the bit array.
        for (j = PACKED_BITS_PER_CHAR - 1; j >= 0; j--)
        {
            mask = 1 << j;
            if (m_checksum & mask)
                *p_bit_array = 1;
            else
                *p_bit_array = 0;

            p_bit_array++;
            *p_reader_array++ = 0; // clear the readers array.
        }
    }

    // The PackedMsg constructor which is the length of a message to be read.

```



```

/////////////////////////////////////////////////////////////////
// READER PARAMETERS STRUCTURES AND CLASSES
/////////////////////////////////////////////////////////////////

// Define a structure which will contain the various Reader parameters.
// The Reader Params class will contain a private copy of this structure.
typedef struct
{
    // User inputs...
    char *input_filename;

    // User provides some combination of following to uniquely locate
    // the registry entry for the signing event...
    UserKey_t user_key;
    time_t date_of_signing;

    char *registry_name; // optional

    // "Super user" inputs, useful for testing and tuning, go here.

    // Non user inputs will go here...

} reader_param_struct;

class ReaderParams
{
public:
    ReaderParams(int argc, char *argv[]); // Constructor for non-gui (cmd line) version

    // Create an accessor which returns a ptr to a const copy of the parameters structure.
    // An alternative is to write accessors for each individual parameter.
    const reader_param_struct *getParams(void) const;

// Private member functions and data structures
private:
    reader_param_struct parameters; // structure containing the user parameters.

    // Function which warns user if parameters are not all present or look incorrect.
    checkParams(void);

},

#endif // PARAMS_H

/////////////////////////////////////////////////////////////////
// paramsdlg.cpp : implementation file
/////////////////////////////////////////////////////////////////
//
#include "stdafx.h"
#include "signer.h"
#include "paramsdlg.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////
// ParamsDlg dialog
/////////////////////////////////////////////////////////////////

ParamsDlg::ParamsDlg(CWnd* pParent /*=NULL*/)
: CDialog(ParamsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(ParamsDlg)
    m_message = L"";
    m_edit_font = (HFONT) 0;
    m_key = 0;
    m_date_size = 0;
    m_date_init_scale = 0.0f;
    //}}AFX_DATA_INIT
}

void ParamsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(ParamsDlg)
    DDX_Text(pDX, IDC_MESSAGE, m_message);
    DDX_MaxChars(pDX, m_message, 256);
}

```



```

long original_ydim,
long x_offset,
long y_offset,
long x_extent,
long y_extent,
int message_length,
unsigned char *key,
long key_length,
/* unused */
char *key_lut,
float *luminance_lut,
float *detail_lut,
unsigned char *thumbnail,
/* if available, use pointer, otherwise NULL */
const unsigned char *referenceBitArray, // bit array ptr: either the known message or estimate.
float *metric, // we will compute a return a crude metric indicating confidence.
unsigned char *message,
int number_channels,
int reading_mode, // output: either 0 or 1, i.e. inefficient but simple */
int bumps, // generally for B&W=1 vs. color == 3
int status = 1;

if(reading_mode == 0) {
    read_8bit_single_channel_OLD_plus_color(
        data, original_xdim, original_ydim, x_offset, y_offset,
        x_extent, y_extent, message_length, key, key_length, key_lut,
        luminance_lut, detail_lut, thumbnail, original_data, referenceBitArray,
        metric, range, message, number_channels, bumps);
}
else if(reading_mode == 1) {
    read_super(
        data, original_xdim, original_ydim, x_offset, y_offset,
        x_extent, y_extent, message_length, key, key_length, key_lut,
        luminance_lut, detail_lut, thumbnail, original_data, referenceBitArray,
        metric, range, message, number_channels, bumps);
}
return(status);
}

// read_8bit_single_channel_OLD_plus_color()
// read_8bit_single_channel_OLD_plus_color()
void read_8bit_single_channel_OLD_plus_color(
    unsigned char *data, // input data to be recognized */
    long original_xdim, // it's x dimension */
    long original_ydim, // it's y dimension */
    long x_offset, // x offset of segment */
    long y_offset, // y offset of segment */
    long x_extent, // x extent of segment */
    long y_extent, // y extent of segment */
    int message_length, // length of message in BITS, also length of message string */
    unsigned char *key, // original 8 bit random key */
    long key_length, // key_length often equal to data_length but not always */
    /* unused */
    char *key_lut, // look up table mapping key value */
    float *luminance_lut, // look up table mapping the signature level to luminance */
    float *detail_lut, // look up table mapping the signature level to local detail */
    unsigned char *thumbnail, // if available, use pointer, otherwise NULL */
    const unsigned char *referenceBitArray, // bit array ptr: either the known message or estimate.
    float *metric, // we will compute a return a crude metric indicating confidence.
    unsigned char *message, // output: either 0 or 1, i.e. inefficient but simple */
    int number_channels,
    int bumps
) {
    unsigned char *pkey, *pdata;
    long i, line, bit;
    int temp, status=1;
    float *key_value = new float[x_extent];
    float *data_float = new float[x_extent];
    float *orig_float = new float[x_extent];
    float *bit_total = new float[message_length];
    //float *bit_mag = new float[message_length];
    float *pkey_value, *pdata_float;

    /* it's y dimension */
    /* x offset of segment */
    /* y offset of segment */
    /* x extent of segment */
    /* y extent of segment */
    /* length of message in BITS, also length of message string */
    /* original 8 bit random key */
    /* key_length often equal to data_length but not always */
    /* look up table mapping key value */
    /* look up table mapping the signature level to luminance */
    /* look up table mapping the signature level to local detail */
    /* if available, use pointer, otherwise NULL */
    const unsigned char *referenceBitArray, // bit array ptr: either the known message or estimate.
    float *metric, // we will compute a return a crude metric indicating confidence.
    unsigned char *message, // output: either 0 or 1, i.e. inefficient but simple */
    int number_channels,
    int bumps
) {
    unsigned char *pkey, *pdata;
    long i, line, bit;
    int temp, status=1;
    float *key_value = new float[x_extent];
    float *data_float = new float[x_extent];
    float *orig_float = new float[x_extent];
    float *bit_total = new float[message_length];
    //float *bit_mag = new float[message_length];
    float *pkey_value, *pdata_float;

```

```

float filter_cf = (float)0.5; // kludge for now
double maxdiff = 40.0; // kludge for now

int key_xlength = 1*(original_xdim-1)/bumps;
for(i=0; i<message_length; i++)
{
    bit_total[i] = (float) 0.0;
    //bit_mag[i] = (float) 0.0;
}

pdata = data;
for(line=y_offset; line<(y_offset+y_extent); line++)
/* FIRST: If either the original image or a thumbnail of the original is available,
then use either a simple or "advanced" dot product to remove it; "advanced" refers
to the idea that you may wish to adjust the gamma to higher order stuff */
float it(pdata, data_float, x_extent, number_channels);
//derivative threshold(data_float, x_extent, number_channels, maxdiff, filter_cf);
//remove_mean(data_float, x_extent);

/* load key values */
int key_offset = (line/bumps)*key_xlength;
pkey = &key[key_offset + x_offset/bumps];
pkey_value = key_value;
if(bumps>1) {
    for(i=x_offset; i<(x_offset+x_extent); i++) {
        if( i%(i+1)*bumps ) pkey++;
    }
}
else {
    for(i=x_offset; i<(x_offset+x_extent); i++) {
        *pkey_value++ = (float) ( (int)key_lut[ (int)*(pkey++) ] );
    }
}
pdata+=(number_channels*x_extent);

/* now step through processed patch and perform simple or "advanced" correlation
detection, keeping the resultant detection values in the accumulators for each bit of the
message_length
bits */
pdata_float = data_float;
pkey_value = key_value;
float running_average = (float) 0.0;
float ftemp;
for (i = 0; i < MOV_AV_KERNEL; i++)
{
    running_average += *(pdata_float++);
}
float mov_av = (float)MOV_AV_KERNEL;
running_average /= mov_av;
pdata_float = data_float;
temp = MOV_AV_KERNEL/2;
int temp1 = temp+1;
if(bumps>1) {
    for (i = x_offset; i < (x_offset + x_extent); i++)
    {
        if (i <= (x_offset + temp) || i >= (x_offset + x_extent - temp) );
        else
        {
            ftemp = *(pdata_float + temp) - *(pdata_float - temp1)) / mov_av;
            running_average += ftemp;
        }
        bit = ( key_offset + i/bumps ) % message_length;
        ftemp = *(pdata_float++) - running_average;
        //bit_mag[bit] += (*pkey_value * *pkey_value);
        bit_total[bit] += ( ftemp * *pkey_value++ );
    }
}
else {
    for (i = x_offset; i < (x_offset + x_extent); i++)
    {
        if (i <= (x_offset + temp) || i >= (x_offset + x_extent - temp) );
        else
        {
            ftemp = *(pdata_float + temp) - *(pdata_float - temp1)) / (float)
            running_average += ftemp;
        }
        bit = ( key_offset + i ) % message_length;
        //bit_mag[bit] += (*pkey_value * *pkey_value);
        bit_total[bit] += ( (pdata_float++) - running_average) * *pkey_value++ );
    }
}
/* time optimized version of above earlier code
int key_foo = key_offset + x_offset;
for(i=x_offset; i<=(x_offset+temp); i++) {

```



```

bit = key_foo++ % message_length;
bit_total[bit] += ( ( *pdata_float++ ) - running_average ) * (pkey_value++);
}

int temp2 = x_offset + x_extent - temp;
float *pdata_float2 = data_float;
float *pdata_float1 = &data_float[temp];
for(i=x_offset+temp+1; i<temp2; i++){
    running_average += ( ( *pdata_float1++ ) - *pdata_float2++ ) / mov_av;
    bit = key_foo++ % message_length;
    bit_total[bit] += ( ( *pdata_float++ ) - running_average ) * (pkey_value++);
}
for(i=0; i<temp; i++){
    bit = key_foo++ % message_length;
    bit_total[bit] += ( ( *pdata_float++ ) - running_average ) * (pkey_value++);
}
}

/* fill the message string based on bit_totals */
for(i=0; i<message_length; i++){
    if (bit_total[i]>0.0)
    {
        message[i]=1;
    }
    else
    {
        message[i]=0;
    }
}

/*
for (i = 0; i < message_length, i++)
{
    // Before normalizing by the magnitudes, be sure we aren't
    // dividing by zero (this happens for an image w/ zero energy.
    if (bit_mag[i] == (float)0.0)
        bit_mag[i] = epsilon;

    bit_total[i] /= (float) sqrt( (double) bit_mag[i] );
}
*/

// Compute the "crude metric", an estimate of rms spread of the
// bit level detector's results. The referenceBitArray is either
// the known message (if it was available to caller) or the
// newly computed estimate of the message.
*metric = get_crude_metric(referenceBitArray, bit_total, range, message_length);

delete () data_float;
delete () orig_float;
delete () bit_total;
delete () key_value;
delete () bit_mag;

return;
}

////////////////////////////////////
float_it()
////////////////////////////////////
void float_it(unsigned char *data, float *data_float,
{
    unsigned char *pdata;
    long i;
    float *pdata;

    pdata = data;
    pdata=data_float;
    if(number_channels == 1){
        for (i = 0; i < x_extent; i++)
            *pdata++ = (float) *pdata++;
    }
    else if (number_channels == 3) {
        for (i = 0; i < x_extent; i++){
            *pdata = (float) *pdata++;
            *pdata += (float) *pdata++;
            *pdata++ += (float) *pdata++;
        }
    }
}
////////////////////////////////////

```

```

// remove_mean()
//
// void remove_mean(float *array, long length)
// {
//     long i;
//     float total = (float) 0.0;
//     for (i = 0; i < length; i++)
//         total += array[i];
//     total /= (float) length;
//     for (i = 0; i < length; i++)
//         array[i] -= total;
// }

////////////////////////////////////
// get_crude_metric()
//
// float get_crude_metric(
//     const unsigned char *actual_message, // the original message, if you have it,
//                                           // otherwise use found message
//     float *bit_total,
//     float *range,
//     int message_length)
// {
//     int i;
//     float avg = (float) 0.0, rms = (float) 0.0, ftemp;
//     *range = (float) 0.0;
//     // add up all the 1's to find an average, as well as 0's
//     for(i=0; i<message_length; i++){
//         if (actual_message[i] > 0)
//             avg += bit_total[i],
//             else
//                 avg -= bit_total[i],
//             avg /= message_length;
//     }
//     // For a zero energy image, avg will equal zero. We replace it
//     // with epsilon.
//     if (avg == 0.0)
//         avg = epsilon;

//     for (i = 0; i < message_length, i++)
//         bit_total[i] /= avg;

//     // now calculate the deviation about the nominal averages
//     for(i=0; i<message_length; i++){
//         if (actual_message[i] > 0)
//             ftemp = bit_total[i] - (float) 1.0;
//             else
//                 ftemp = bit_total[i] + (float) 1.0;
//         if ( fabs( (double)ftemp ) > (double) *range)
//             *range = (float) fabs( (double) ftemp);
//         rms += (ftemp * ftemp);
//     }
//     ftemp = rms/ ((float)message_length - (float) 1.0);
//     rms = (float) sqrt(ftemp);
//     return( rms); // returns crude spread metric */
// }

int derivative_threshold(float *data, long length, int number_channels, double maxdiff, float
filter_cf)
{
    long i;
    int status = 1;
    float *pdata, llast, last;
    double diff;
    float replacement = (float)0.0;
    if(number_channels == 3)maxdiff *= 3.0;
    last = llast = data[0];
    pdata = &data[1];
}

```

```

for(i=1; i<length; i++){
    diff = (double)*pdata - last;
    last = *pdata;
    if( fabs(diff) > maxdiff ){
        if ( diff > 0.0 ) diff = replacement;
        else diff = -replacement;
    }
    *pdata = last + (float)diff;
    last = *pdata++;
}

return(status);
}

void read_super(
    unsigned char *data,
    long original_xdim,
    long original_ydim,
    long *offset,
    long *offset,
    long *extent,
    long *extent,
    long *message_length,
    unsigned char *key,
    long key_length,
    /* unused */
    char *key_lut,
    float *luminance_lut,
    float *detail_lut,
    unsigned char *thumbnail,
    unsigned char *original_data,
    /* if available, use pointer, otherwise NULL */
    /* if available, use pointer, otherwise NULL */
    const unsigned char *reference_bitarray, // bit array ptr: either the known message or estimate.
    float *metric, // we will compute a return a crude metric indicating confidence.
    float *range,
    unsigned char *message,
    int number_channels,
    int bumps
){
    unsigned char *pkey, *pdata;
    long i, line, bit;
    int status=1, bits, fftdim, j, highest;
    float *bit_total = new float(message_length);
    float *bit_mag = new float(message_length);
    float *key_value = new float(x_extent)*pkey_value;
    int key_xlength = 1*(original_xdim-1)/bumps;

    for(i=0; i<message_length; i++){
        bit_total[i] = (float) 0.0;
        bit_mag[i] = (float) 0.0;
    }

    // find power of 2 higher than highest dimension
    if(x_extent > y_extent) highest = x_extent;
    else highest = y_extent;
    bits = 1 + (int){ log( (double)highest - 0.5 ) / log(2.0) };
    fftdim = (int)pow(2.0, (double)bits + 0.00000001);

    // create array
    float *image = new float[fttdim*(fttdim+2)];
    float *wr = new float[fttdim];
    float *wi = new float[fttdim];
    float *pimage;
    pimage = image;
    for(i=0; i<(fttdim+2); i++){
        *pimage++ = (float)*pdata++;
        total += *pimage++;
    }

    // convert either a B&W image or a color image to a single floating point luminance image
    float total;
    if(number_channels == 1){
        pdata = data;
        for(i=0; i<y_extent; i++){
            pimage = &image(i*(fttdim+2));
            for(j=0; j<x_extent; j++){
                *pimage = (float)*pdata++;
                total += *pimage++;
            }
        }
    }
    else if(number_channels == 3){
        pdata = data;
        for(i=0; i<y_extent; i++){
            pimage = &image(i*(fttdim+2));
            for(j=0; j<x_extent; j++){
                *pimage++ = (float)*pdata++;
                total += *pimage++;
            }
        }
    }
}

```

```

    *pimage = (float)*pdata++;
    *pimage = (float)*pdata++;
    *pimage = (float)*pdata++;
    total += *pimage++;
}

// weird derivative threshold
int choo=0;
if(choo){
    // remove dc
    total /= ((float)y_extent * (float)x_extent);
    for(i=0; i<y_extent; i++){
        pimage = &image(i*(fttdim+2));
        for(j=0; j<x_extent; j++){
            *pimage++ -= total;
        }
    }

    float *detail_vector;
    float *detail_vector = new float(x_extent);
    int start = 5;
    int stop = 500;
    float scale = (float)0.5;
    for(i=0; i<y_extent; i++){
        get_read_detail_vector(detail_vector, data, x_extent, i, y_extent, number_channels, start, stop, scale,
            , image, fftdim);
        detail_vector = detail_vector;
        pimage = &image(i*(fttdim+2));
        for(j=0; j<x_extent; j++){
            *pimage++ += *pdetail_vector++;
        }
        delete [] detail_vector;
    }

    //float filter_cf = (float)0.5; // kludge for now
    //double maxdiff = 40.0; // kludge for now
    //for(line=0; line<y_extent; line++){
    //    derivative_threshold(&image[line*(fttdim+2)], x_extent, 1, maxdiff, filter_cf);
    //}

    // easy does the window ??
    // for now multiply the last four values near the edges by a linear ramp to zero, simply
    // to avoid total edge weirdness
    int window=0;
    if(window != 0){
        if(x_extent > 10 && y_extent > 10){
            float mult[4];
            mult[0] = (float)0.2; mult[1] = (float)0.4; mult[2] = (float)0.6; mult[3] = (float)0.8;
            pmult = mult;
            for(i=1; i<5; i++){
                pimage = &image(i*(fttdim+2));
                for(j=0; j<x_extent; j++){
                    *pimage++ *= *pmult;
                    pmult++;
                }
            }
            for(i=1; i<5; i++){
                pimage = &image(y_extent - i*(fttdim+2));
                for(j=0; j<x_extent; j++){
                    *pimage++ *= *pmult;
                    pmult++;
                }
            }
            for(i=0; i<y_extent; i++){
                pimage = &image(i*(fttdim+2));
                pmult = mult;
                for(j=1; j<5; j++){
                    *pimage++ *= *pmult;
                    pimage = &image(y_extent - i*(fttdim+2));
                    for(j=0; j<x_extent; j++){
                        *pimage++ *= *pmult;
                        pmult++;
                    }
                }
            }
        }
    }

    // fft arrays
    realfft2d_in_place(image, bits, 0, wr, wi);
    // filter them
    // phase difference only to start
    // calculate phase differences and reload them into real and imaginary1 *
    float mag1, preall, *pimaginary1;
    // double power = 0.8;
    preall = image; pimaginary1 = &image[fttdim];
    for(i=0; i<(i*(fttdim+2)); i++){
        mag1 = (float)fabs( (double)*preall ) + (float)fabs( (double)*pimaginary1 );
        if(mag1 == (float)0.0){
            *preall++ = (float)0.0;
        }
    }
}

```

```

    }
    else {
        //mag1 = (float)pow((double)mag1,power);
        //preali++ /= mag1;
        *(pimaginary1++) /= mag1;
    }
}
preali+=fftdim;
pimaginary1+=fftdim;
}

// remove low and/or high frequencies
// the DC should reside at row one, fftdim/2
int mo0 = 0;
if(mo0) {
    int low = 1;
    int xcount=low*2-1;
    pimage = image[(fftdim/2) - low +1];
    for(i=0,i<2*low;i++) {
        for(j=0;j<xcount;j++)*(pimage++) = (float)0.0;
        pimage += (fftdim - xcount);
    }
}

// inverse fft
realfft2d_in_place(image,bits,1,wr,wl);
for(line=y_offset, line<(y_offset+y_extent); line++) {
    // load key values */
    pkey = key[(line/bumps) * key_xlength + x_offset/bumps];
    for(i=x_offset;x_offset<x_extent;i++) {
        //y_value[i-x_offset] = (float){ (int)key_lut[ (int)pkey ] };
        if( (i+1)/bumps != pkey++ );
    }

    /* now step through processed patch and perform simple or "advanced" correlation detection,
    keeping the resultant detection values in the accumulators for each bit of the
    message_length */
    bits = 0;
    pimage = image[(line-y_offset)*fftdim];
    pkey_value = key_value;
    for(i=x_offset;x_offset<x_extent;i++) {
        bit = ( (line/bumps)*key_xlength + i/bumps ) & message_length;
        bit_mag[bit] = (pkey_value * pkey_value);
        bit_total[bit] += ( *pimage++ ) * *pkey_value++;
    }

    /* fill the message string based on bit_totals */
    for(i=0, i<message_length; i++) {
        if(bit_total[i]>0.0)
            message[i]=1;
        else
            message[i]=0;
    }

    for( i = 0; i < message_length; i++)
        // Before normalizing by the magnitudes, be sure we aren't
        // dividing by zero (this happens for an image w/ zero energy.
        if (bit_mag[i] == (float)0.0)
            bit_mag[i] = epsilon;

        bit_total[i] /= (float) sqrt( (double) bit_mag[i] );

        // Compute the "crude metric", an estimate of rms spread of the
        // bit level detector's results. The referenceBitArray is either
        // the known message (if it was available to caller) or the
        // newly computed estimate of the message.

        *metric = get_crude_metric(referenceBitArray, bit_total, range, message_length);

        delete [] bit_total;
        delete [] bit_mag;
        delete [] key_value;
        delete [] image;
        delete [] wr;
        delete [] wl;
    }
}

return;
}

// get_read_detail_vector()
// =====
//
// int get_read_detail_vector(
//     unsigned char *data,
//     int xdim,
//     int total_rows,
//     int number_channels,
//     int start,
//     int stop,
//     float scale,
//     float *image,
//     int fftdim
// ) {
//     unsigned char *p1;
//     float *pdata,*p2;
//     int i;
//     float base,temp;
//     float *pdetail_vector=detail_vector;
//
//     // this function creates a "scaling" vector for the current scan line,
//     // based on a crude metric of "local detail"
//     if(number_channels == 1) {
//     }
//     else if(number_channels == 3) {
//         pdata = image[row*fftdim];
//         if(row == 0)p1 = pdata[3*row*xdim];
//         else p1 = pdata[3*(row-1)*xdim];
//         if(row == (total_rows-1))p2 = image[row*fftdim];
//         else p2 = image[(row+1)*fftdim];
//         // perform first and last elements outside loop so that an internal if statement is
//         // avoided
//         base = (float)*(p1++),base+=(float)*(p1++),base+=(float)*(p1++);
//         base+= *(p2++);
//         base+= (float)2.0 * *(pdata+1);
//         temp = base/(float)4.0 * *(pdata++);
//         float denom = (float)(stop-start)/((float)1.0-scale);
//         float mult;
//         base = (float)fabs( (double)temp );
//         if( base > (float)start ) {
//             if(base > (float)stop)mult = (float)1.0 - scale;
//             else mult = (base - (float)start)/denom;
//             *pdetail_vector++ = mult * temp;
//         }
//         else *pdetail_vector++ = (float)0.0;
//         for(i=1;i<(xdim-1);i++) {
//             base = (float)*(p1++),base+=(float)*(p1++),base+=(float)*(p1++);
//             base+= *(p2++);
//             base+= *(pdata+1);
//             base+= *(pdata-1);
//             temp = base/(float)4.0 * *(pdata++);
//             base = (float)fabs( (double)temp );
//             if( base > (float)start ) {
//                 if(base > (float)stop)mult = (float)1.0 - scale;
//                 else mult = (base - (float)start)/denom;
//                 *pdetail_vector++ = mult * temp;
//             }
//             else *pdetail_vector++ = (float)0.0;
//         }
//         base = (float)*(p1++),base+=(float)*(p1++),base+=(float)*(p1++);
//         base+= *(p2);
//         base+= (float)2.0 * *(pdata-1);
//         temp = base/(float)4.0 * *(pdata);
//         base = (float)fabs( (double)temp );
//         if( base > (float)start ) {
//             if(base > (float)stop)mult = (float)1.0 - scale;
//             else mult = (base - (float)start)/denom;
//             *pdetail_vector = mult * temp;
//         }
//         else *pdetail_vector = (float)0.0;
//     }
//     return(i);
// }

// =====
//
// Header file for the Reader core algorithm functions.
// =====

```

```

unsigned char *thumbnail, /* if available, use pointer, otherwise NULL */
unsigned char *original_data, /* if available, use pointer, otherwise NULL */

const unsigned char *referenceBitArray, /* bit array ptr: either the known message or
estimate.
float *metric, // we will compute a return a crude metric indicating
confidence.
float *range,
unsigned char *message,
int number_channels,
int bumps);

int get_read_detail_vector(
float *detail_vector,
unsigned char *data,
int xdim,
int ydim,
int row,
int total_rows,
int number_channels,
int start,
int stop,
float scale,
float *image,
int ffdim
);

#endif // READ_H

// readdlg.cpp : implementation file
//
#include "stdafx.h"
#include "signer.h"
#include "readdlg.h"

#ifdef _DEBUG
#define THIS_FILE static char BASED_CODE THIS_FILE[] = __FILE__
#endif

// ReadDlg dialog
//
// ReadDlg()
//
// Constructor for the Reader Parameters Dialog object. A ReadDlg
// object is created to manage a dialog in which the user is able
// to set the parameters used by the Reader and associated core
// algorithms.
//
// ReadDlg::ReadDlg(CWnd* pParent /*=NULL*/)
// : CDialog(ReadDlg::IDD, pParent)
// {
//     ((AFX_DATA_INIT(ReadDlg)
//     m_user_key = 0;
//     m_msg_length = 0;
//     m_gain = (float) 0.0;
//     m_bump_size = 0;
//     m_detail_lut_scale = 0.0f;
//     ))AFX_DATA_INIT
// }

void ReadDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //((AFX_DATA_MAP(ReadDlg)
    DDX_Text(pDX, IDC_READ_KEY, m_user_key);
    DDX_MinMaxInt(pDX, m_msg_length, 0, 65535);
    DDX_Text(pDX, IDC_READ_LENGTH, m_msg_length);
    DDX_MinMaxInt(pDX, m_gain, 1, 65535);
    DDX_Text(pDX, IDC_READ_GAIN, m_gain);
    DDX_MinMaxInt(pDX, m_bump_size, 1, 256);
    DDX_Text(pDX, IDC_READ_SCALE, m_detail_lut_scale);
    DDX_MinMaxInt(pDX, m_bump_size, 1, 256);
    DDX_Text(pDX, IDC_DETAIL_LUT_SCALE, m_detail_lut_scale);
    DDX_MinMaxInt(pDX, m_detail_lut_scale, 1.0e-003f, 1.0e+006f);
    //))AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(ReadDlg, CDialog)
//{{AFX_MSG_MAP(ReadDlg)
//{{AFX_MSG_MAP(ReadDlg)
END_MESSAGE_MAP()

```

```

////////////////////////////////////
// ReadDlg message handlers
void ReadDlg::OnOK()
{
    CDialog::OnOK();
}

////////////////////////////////////
// readdlg.h : header file
//
// ReadDlg dialog
class ReadDlg : public CDialog
{
// Construction
public:
    ReadDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    enum { IDD = IDD_READ_DIALOG };
    UINT m_user_key;
    UINT m_msg_length;
    float m_gain;
    int m_bump;
    float m_detail_lut_scale;
    float m_detail_lut_scale;
    ///////////////////////////////////
// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

// Generated message map functions
//{{AFX_MSG(ReadDlg)
    virtual void OnOK();
//}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

READDLG.H

```

////////////////////////////////////
// readdlg.h : header file
//
// ReadDlg dialog
class ReadDlg : public CDialog
{
// Construction
public:
    ReadDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    enum { IDD = IDD_READ_DIALOG };
    UINT m_user_key;
    UINT m_msg_length;
    float m_gain;
    int m_bump;
    float m_detail_lut_scale;
    float m_detail_lut_scale;
    ///////////////////////////////////
// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

// Generated message map functions
//{{AFX_MSG(ReadDlg)
    virtual void OnOK();
//}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

RESOURCE.H

```

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Signer.rc

```

```

#define IDR_MAINFRAME          2
#define IDR_DIBTYPE           3
#define IDD_ABOUTBOX          100
#define IDC_MESSAGE            101
#define IDD_PARAMS_DIALOG     101
#define IDC_GAIN               102
#define IDC_GAIN_LABEL         103
#define IDD_READ_DIALOG       103
#define IDC_MESSAGE_LABEL      104
#define IDC_EDIT_GAIN          106
#define IDC_EDIT_GAMMA         107
#define IDC_EDIT_KEY           108
#define IDC_READ_KEY           110
#define IDC_READ_LENGTH        111
#define IDC_READ_LENGTH        111
#define IDC_READ_LENGTH        112
#define IDC_READ_LENGTH        112
#define IDC_READ_LENGTH        113
#define IDC_READ_LENGTH        113
#define IDC_READ_LENGTH        114
#define IDC_READ_LENGTH        115
#define IDC_READ_LENGTH        115
#define IDC_READ_LENGTH        116
#define IDC_READ_LENGTH        117
#define IDC_READ_LENGTH        118
#define IDC_READ_LENGTH        119
#define IDC_READ_LENGTH        120
#define IDC_READ_LENGTH        121
#define IDC_READ_LENGTH        121
#define IDC_READ_LENGTH        122
#define IDC_READ_LENGTH        123
#define IDC_READ_LENGTH        124
#define IDC_READ_LENGTH        125
#define IDC_READ_LENGTH        126
#define IDC_READ_LENGTH        127
#define IDC_READ_LENGTH        128
#define IDC_READ_LENGTH        129
#define IDC_READ_LENGTH        130
#define IDC_READ_LENGTH        131
#define IDC_READ_LENGTH        132
#define IDC_READ_LENGTH        133
#define IDC_READ_LENGTH        134
#define IDC_READ_LENGTH        135
#define IDC_READ_LENGTH        136
#define IDC_READ_LENGTH        137
#define IDC_READ_LENGTH        138
#define IDC_READ_LENGTH        139
#define IDC_READ_LENGTH        140
#define IDC_READ_LENGTH        141
#define IDC_READ_LENGTH        142
#define IDC_READ_LENGTH        143
#define IDC_READ_LENGTH        144
#define IDC_READ_LENGTH        145
#define IDC_READ_LENGTH        146
#define IDC_READ_LENGTH        147
#define IDC_READ_LENGTH        148
#define IDC_READ_LENGTH        149
#define IDC_READ_LENGTH        150

```

```

// Next default values for new objects
//

```

```

// #define APSTUDIO_INVOKED
// #define APSTUDIO_READONLY_SYMBOLS
// #define _APS_NEXT_RESOURCE_VALUE 106
// #define _APS_NEXT_COMMAND_VALUE 32764
// #define _APS_NEXT_CONTROL_VALUE 122
// #define _APS_NEXT_SYMED_VALUE 102
// #endif

```

SIGN.CPP

```

////////////////////////////////////
// FILE: Sign.cpp
//
// DESCRIPTION:
// Core signing functions of the digimarc technology.
// Created July 1995.
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
//
// #include "sign.h"
// #include <math.h>
// #include <stdio.h>
//
// * this function loads the scaling factor based on luminance */
int load_luminance_lut( float *luminance_lut, float gamma) // explicitly written for 8 bit
{
    int i,status=1;

    luminance_lut[0] = (float) 0.; /* don't put any signature energy into zero luminance
    (black) */
    for(i=1; i<256; i++)
    {
        luminance_lut[i] = (float) pow((double)i, (double) gamma);
    }
    return(status);
}

////////////////////////////////////
// load_key_lut()
//
// This function just assigns mainly 0's, 1's, -1's, 2's and -2's
// to the key values, scaled by the scale point.
// scale point is a simple integer between 1 and 127
// about 30 to 50 should be about right for first tests
//
// float load_key_lut(char *key_lut, float gain)
//
// int i,base_gain,ifraction;
// float rms,fraction;
// gain /= (float)100.0;
// rms = gain;
//
// base_gain = (int)gain;
// ifraction = gain - (float)base_gain;
// ifraction = (int)((float) 127.0 * fraction );
// if(ifraction == 0){
//     for(i=0;i<128;i++)key_lut[i]=(char)base_gain;
//     for(i=0;i<128;i++)key_lut[i+128]=-(char)base_gain;
// }
// else {
//     for(i=0;i<(128-ifraction);i++){
//         key_lut[i]=(char)base_gain;
//         key_lut[i+128]=-(char)base_gain;
//     }
//     for(i=(128-ifraction);i<128;i++){
//         key_lut[i]=(char)(base_gain+1);
//         key_lut[i+128]=-(char)(base_gain+1);
//     }
// }
// return( rms);
//
//
// The following functions are core algorithms which include
// 1) additional capabilities for signing Color images, and
// 2)
//

```

```

scale*=DETAIL_NORMALIZER;
for(i=0;i<DETAIL_START;i++)detail_lut[i]=(float)1.0;
for(i=DETAIL_START;i<DETAIL_STOP;i++)
{
    detail_lut[i] = (float)1.0 + scale*((float)(i-DETAIL_START)/length);
}
for(i=DETAIL_STOP;i<DETAIL_TOTAL;i++)detail_lut[i]=detail_lut[DETAIL_STOP-1];

return(status);
}

// sign_8bit_single_channel_or_color()
// written for the march 1996 bump incarnation
// int sign_8bit_single_channel_or_color()
// unsigned char *data, // input data to be signed
// long data_length, // it's length
// long xdim, // it's x dimension
// long ydim, // it's y dimension
// unsigned char *message, // either 0 or 1, i.e. inefficient but simple
// int message_length, // length of message in BITS, also length of message string
// unsigned char *key, // 8 bit random key, uniformly distributed
// long key_length, // key_length often equal to data_length but not always
// *unused,
// char *key_lut, // look up table mapping key value
// float *luminance_lut, // look up table mapping the scaling to luminance values
// float *detail_lut, // look up table mapping the scaling to luminance values
// int signing_mode, // current options: STANBARD, STRICT_LUMINANCE
// unsigned char *data_out, // signed output data in same length and format as input
// int number_channels, // added in late february 1996 to begin work on 3 color 24 bit
// color images
// int bumps // added in March 1996 to implement bumps
}

// unsigned char *pdata;
// unsigned char *p_out;
// unsigned char *pkey;
// long i;
// int j,k;
// int lum_change,status=1;
// float *temp,delta;
// float *detail_vector = new float[xdim];
// float *pdetail_vector,local_gain;
// int key_xlength;
// key_xlength = 1+(xdim-1)/bumps;

if(number_channels == 1){
    pdata = data;
    p_out = data_out;
    for(i=0;i<ydim;i++){
        // load local detail values for this row
        get_detail_vector(detail_vector,pdata,xdim,i,ydim,detail_lut,number_channels);
        pdetail_vector = detail_vector;
        pkey=key[(i/bumps)*key_xlength];
        pmessage = pmessage[(i/bumps)*key_xlength];
        for(j=0;j<xdim;j++){
            lum_change = key_lut[(int)*pkey];
            if(lum_change == 0){
                *p_out++ = *pdata++;
            }
            else {
                pdetail_vector++;
            }
        }
        else {
            local_gain = *(pdetail_vector++) * luminance_lut[*pdata];
            if( abs(lum_change) > 1 ){ // this is the anti-sparkles check
                if( local_gain > (float)3.5 ){
                    if(lum_change > 0)lum_change = 1;
                    else lum_change = -1;
                }
            }
            delta = (float)lum_change * local_gain;
            if( !(*pmessage) )
                delta = -delta; // invert current snowy image luminance value ... key

            ftemp = (float)*pdata++ + delta;
            if(ftemp > (float)255.0)*p_out++ = (unsigned char)255;
            else if(ftemp<(float)0.0)*p_out++ = (unsigned char)0;
            else *p_out++ = (unsigned char)(ftemp+(float)0.5);
        }
    }
    if( (j+1)*bumps == 0 ){

```



```

// Set pointer to the DIB of the image which is to be saved.
if (view_type == ORIGINAL_VIEW)
    hSavedDIB = m_hOriginalDIB;
else if (view_type == SIGNED_VIEW)
    hSavedDIB = m_hSignedDIB;
else if (view_type == ALIGNED_VIEW)
    hSavedDIB = m_hAlignedImage->GetHDIIB();
else if (view_type == STATUS_VIEW)
{
    // This is the unusual case where we are not saving a DIB.
    // Instead, we write out the character strings of the status view.
    file.Close(); // close the binary file, create ostream instead
    ofstream of(pszPathName); // Text output file stream
    CDibView stat_stream; // For in-memory formatting of the string
    CDibView stat_view;
    stat_view = GetActiveView();
    stat_view->createStatusStream(stat_stream);
    // Write the status information to the file
    of << stat_stream.str();
    of.close();
    delete stat_stream.str(); // Once we use .str, we have to delete it.
    return TRUE;
}

TRY
{
    BeginWaitCursor();
    bSuccess = ::SaveDIB(hSavedDIB, file);
    file.Close();
}
CATCH (CException, eSave)
{
    file.Abort(); // will not throw an exception
    EndWaitCursor();
    ReportSaveLoadException(pszPathName, eSave,
        TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);
    return FALSE;
}
END_CATCH

EndWaitCursor();
SetModifiedFlag(FALSE); // back to unmodified
if (bSuccess)
{
    // may be other-style DIB (load supported but not save)
    // or other problem in SaveDIB
    MessageBox(NULL, "Couldn't save DIB", NULL,
        MB_ICONINFORMATION | MB_OK);
}
if (m_state == IMAGE_SIGNED_AND_VERIFIED)
    m_state = IMAGE_SIGNED_AND_SAVED;
// Save the name of the saved file.
m_filename = pszPathName;
// If the user switch is set, create a "Status view" (iff it doesn't
// already exist), and print it.
if (m_autoprint)
{
    CDibView *p_status_view;
    p_status_view = (CDibView*) CreateUniqueView(STATUS_VIEW);
    p_status_view->OnFilePrint();
}
else
    UpdateAllViews(NULL); // if status view present, needs update
return bSuccess;
}

void CDibDoc::ReplaceHDIIB(HDIIB hDIB)
{
    if (m_hOriginalDIB != NULL)
    {
        ::GlobalFree((HGLOBAL) m_hOriginalDIB);
        m_hOriginalDIB = hDIB;
    }
}

// CDibDoc diagnostics
// =====
#ifdef _DEBUG
void CDibDoc::AssertValid() const
{
    CDocument::AssertValid();
}
#endif

```

```

void CDibDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}

//end of //DEBUG

////////////////////////////////////
// DumpBitmapInfoHeader()
// Diagnostic tool which dumps out some information about the DIB's
// header. Only used for test/debug purposes.
//end of //DEBUG
void CDibDoc::DumpBitmapInfoHeader() const
{
    int
    long
    LPSTR
    LPBITMAPINFOHEADER
    LPBITMAPINFO
    LPbmi;

    int i, cxDIB, cyDIB;
    long num_pixels, num_colors;
    LPSTR lpDIB;
    LPBITMAPINFOHEADER lpDIBHdr;
    LPBITMAPINFO lpbmi;

    HDB hOriginalDIB = GetOriginalHDB();
    if (hOriginalDIB == NULL)
        return;

    // Lock the DIB in memory
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hOriginalDIB);

    // Get ptr to the dib header space.
    lpDIBHdr = (LPBITMAPINFOHEADER) lpDIB;

    // get pointer to BITMAPINFO (Min 3.0)
    lpbmi = (LPBITMAPINFO) lpDIB;
    RGBQUAD *bmiColors = lpbmi->bmiColors;

    cxDIB = (int) ::DIBWidth(lpDIB); // X size of DIB
    cyDIB = (int) ::DIBHeight(lpDIB); // Y size of DIB
    num_pixels = (long) cxDIB * cyDIB;
    num_colors = ::DIBNumColors(lpDIB);

    if (lpDIBHdr->biCompression != 0)
    {
        TRACE("Can't cope with compressed image (compression = %d)\n",
              lpDIBHdr->biCompression);
        ::GlobalUnlock((HGLOBAL) m_hOriginalDIB);
        return;
    }

    TRACE("BITMAPINFOHEADER contents are:\n");
    TRACE("HeaderSize = %d, width = %d, height = %d, num_pixels = %d\n",
          lpDIBHdr->biSize, cxDIB, cyDIB, num_pixels);
    TRACE("Planes = %d bits per pixel = %d\n",
          lpDIBHdr->biPlanes, lpDIBHdr->biBitCount);
    TRACE("CompressionMethod = %d\n", lpDIBHdr->biCompression);
    TRACE("SizeOfBitmap = %d\n", lpDIBHdr->biSizeImage);
    TRACE("num_colors = %d\n", num_colors);
    TRACE("HorResolution = %d, VertResolution = %d\n",
          lpDIBHdr->biXPerMeter, lpDIBHdr->biYPerMeter);
    TRACE("bmiColorsUsed = %d numSignColors = %d\n",
          lpDIBHdr->biClrUsed, lpDIBHdr->biClrImportant);

    // Dump the palette. This is only for severe debugging situations.
    TRACE("The contents of the palette:\n");
    for (i = 0; i < num_colors; i++)
    {
        TRACE("%d %2x %2x %2x\n", i,
              (int) bmiColors->rgbRed, (int) bmiColors->rgbGreen,
              (int) bmiColors->rgbBlue);
        bmiColors++;
    }

    // We are now all done w/ the Original DIB. Unlock it.
    ::GlobalUnlock((HGLOBAL) hOriginalDIB);
}

////////////////////////////////////
// Member function which
// builds a snow image in place.
//end of //DEBUG

typedef char *HPSTR; // huge pointer to a string NOW OBSOLETE

```

```

// MakeSnow()
// Creates a snow image, and sets the member variable m_hSnowyDIB, which
// is a DIB handle to the new snow image DIB. The snow image which is
// created is sized based on the parent DIB handle passed in, and it
// has all the same bits, header and palette stuff.
//end of //DEBUG
void CDibDoc::MakeSnow(HDB hParentDIB)
{
    int
    long
    cxDIB, cyDIB;
    num_pixels, num_colors;
    total_size, image_byte;
    LPSTR lpDIB, lpSnowyDIB; // Pointer to BITMAPINFOHEADER
    LPBITMAPINFOHEADER lpSnowyDIBHdr;
    HPSTR hPSTR
    hPSTR // Huge ptrs for copying the image.

    // HDB hOriginalDIB = GetOriginalHDB();
    if (hParentDIB == NULL)
        return;

    // Get the size of the parent DIB
    total_size = GlobalSize((HGLOBAL) hParentDIB);

    // Create space for the snow image (on 1st call only).
    if (m_hSnowyDIB == NULL)
    {
        m_hSnowyDIB = (HDB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, total_size);
        if (m_hSnowyDIB == 0)
        {
            MessageBox(NULL,
                      "Insufficient memory is available for the 'snowy image'",
                      "Digimarc Signer Warning",
                      MB_ICONINFORMATION | MB_OK);
            return;
        }
    }

    // Lock the two DIBs in memory
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hParentDIB);
    lpSnowyDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hSnowyDIB);

    src_data = (char *) lpDIB;
    dest_data = (char *) lpSnowyDIB;

    // Copy the BITMAPINFOHEADER, palette, and actual image byte data by byte.
    {
        *dest_data++ = *src_data++;
    }

    // For debug: reset the pointers.
    src_data = (char *) lpDIB;
    dest_data = (char *) lpSnowyDIB;
    if (*src_data != *dest_data)
        TRACE("DEBUG: after copy into snowy image, 1st chars aren't equal\n");

    // We are now all done w/ the Parent DIB. Unlock it.
    ::GlobalUnlock((HGLOBAL) hParentDIB);

    // Get ptr to the snowy dib header space.
    lpSnowyDIBHdr = (LPBITMAPINFOHEADER) lpSnowyDIB;

    hPSTR hPSTR = ::FindDIBBits(lpSnowyDIB);

    cxDIB = (int) ::DIBWidth(lpSnowyDIB); // X size of DIB
    cyDIB = (int) ::DIBHeight(lpSnowyDIB); // Y size of DIB
    num_pixels = (long) cxDIB * cyDIB;
    num_colors = ::DIBNumColors(lpSnowyDIB);

    if (lpSnowyDIBHdr->biCompression != 0)
    {
        TRACE("Can't cope with compressed image (compression = %d)\n",
              lpSnowyDIBHdr->biCompression);
        ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
        return;
    }

    TRACE("width = %d, height = %d, num_pixels = %d\n", cxDIB, cyDIB, num_pixels);
    TRACE("num_colors = %d\n", num_colors);

    if (m_bBitsPerPixel != 8 && m_bBitsPerPixel != 24)
    {

```

```

    TRACE("At this time, only build snowy image for 8 or 24 bit images\n");
    ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
    return;
}

if (m_BitsPerPixel == 8 || m_BitsPerPixel == 24)
{
    CoKey coKey(m_Params->GetKey(), (BITMAPINFO *) lpSnowyDIBHdr,
        hpSnowyDIBBits);
    ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
}

// Sign()
// This is the function which calls upon the core signing algorithms.
// WARNING: CURRENTLY THIS FUNCTION ASSUMES THAT WE ALWAYS ARE SIGNING
// THE "ORIGINAL IMAGE" DIB. THIS MAY BE A BUG.
// First shot at a function which calls the signer core algorithms
void CDibDoc::Sign(void)
{
    long num_pixels, num_colors;
    DWORD image_byte;
    HPSTR src_data, dest_data; // Huge ptrs for copying the image.
    float rms;
    int num_channels;

    HBIT hOriginalDIB = GetOriginalHBIT();
    return;

    // Create space for the signed image DIB.
    m_hSignedDIB = (HBIT) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, m_dwTotalDIBSize);
    if (m_hSignedDIB == 0)
    {
        MessageBox(NULL,
            "Insufficient memory is available for the signed image",
            "Digimarc Signer Warning",
            MB_ICONINFORMATION | MB_OK);
        return;
    }

    // Create image objects for the images. Note that this locks them in memory.
    Image snowyImage(m_hSnowyDIB);
    Image unsignedImage(m_hOriginalDIB);

    // This is ugly, but I have to copy the DIB header stuff into the signed DIB
    // before I can create the signed image object.
    dest_data = (char *) ::GlobalLock((HGLOBAL) m_hSignedDIB);

    // We want to copy the BITMAPINFO structure from the unsigned to the signed DIB
    src_data = unsignedImage.GetUpDIB();

    // Copy the BITMAPINFOHEADER and palette to the signed DIB space, byte by byte.
    for (image_byte = 0; image_byte < unsignedImage.GetSizeofHeader(); image_byte++)
    {
        *dest_data++ = *src_data++;
    }

    ::GlobalUnlock((HGLOBAL) m_hSignedDIB);

    // Now create the signed image object, which will lock the DIB in memory again.
    Image signedImage(m_hSignedDIB);

    // For each, create a "byte-wise" packed data array from the DIB 4-byte packing
    snowyImage.MakePackedData(FORCE_TO_1_CHANNEL); // snowy image always 1 chan
    unsignedImage.MakePackedData();
    signedImage.MakePackedData();

    num_pixels = (long) unsignedImage.GetXDIM() * unsignedImage.GetYDIM();
    num_colors = unsignedImage.GetNumColors();

    if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)
    {
        TRACE("At this time, only sign 8 and 24 bit images\n");
        return;
    }

    // Create and load the luminance scaling look up table.
    float *luminance_lut = new float[256];
    ::load_luminance_lut(luminance_lut, m_Params->GetGamma());

    // Create and load the key look up table.
    char *key_lut = new char[256];
    rms = ::load_key_lut(key_lut, m_Params->GetGain());

    long data_length = unsignedImage.GetXDIM() * unsignedImage.GetYDIM();

    // Create a packed msg (will be a user input in future).
    if (m_PackedMsg != NULL)
        delete m_PackedMsg;
    m_PackedMsg = new PackedMsg( (const char *) m_Params->GetMessage());

    // Set up some arguments and call the core signer.
    int x_dim = unsignedImage.GetXDIM();
    int y_dim = unsignedImage.GetYDIM();

    if (unsignedImage.GetBitsPerPixel() == 8)
    {
        num_channels = 1;
    }
    else if (unsignedImage.GetBitsPerPixel() == 24)
    {
        num_channels = 3;
    }

    // const float lut_scale = (float)1.0; // Later this will be user controlled.
    float *detail_lut = new float[DETAIL_TOTAL];
    ::load_detail_lut(detail_lut, m_Params->GetLutScale());

    ::sign_8bit_single_channel_or_color(unsignedImage.GetPackedData(),
        data_length,
        x_dim,
        y_dim,
        m_PackedMsg->GetMsgBitArray(),
        m_PackedMsg->GetMsgBitArrayLength(),
        snowyImage.GetPackedData(),
        data_length,
        key_lut,
        luminance_lut,
        detail_lut,
        STANDARD,
        signedImage.GetPackedData(),
        num_channels,
        m_Params->GetBumpSize());

    delete [] detail_lut;

    // Set the timestamp indicating when we signed this puppy.
    m_Params->UpdateSignatureTime();

    delete [] luminance_lut;
    delete [] key_lut;

    // Now unpack the data in the image object, back into the standard DIB format
    signedImage.UnpackData();
}

// Read()
// The read function is the interface to the core recognition algorithm.
// It sets up the necessary data structures needed by the core routine
// and makes the call.
// void CDibDoc::Read(HBIT hSignedDIB, BOOL use_super_reader)
{
    long num_pixels, num_colors;
    int num_channels;
    int reading_mode;

    // Create image objects for the images. Note that this locks them in memory.
    Image snowyImage(m_hSnowyDIB);
    Image signedImage(hSignedDIB);

    // Create a "byte-wise" packed data array from the DIB 4-byte packing
    signedImage.MakePackedData();
    snowyImage.MakePackedData(FORCE_TO_1_CHANNEL); // Snowy images always 1 ch.
    // unsignedImage.MakePackedData();

    num_pixels = (long) signedImage.GetXDIM() * signedImage.GetYDIM();
    num_colors = signedImage.GetNumColors();

    if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)

```

```

TRACE("At this time, only recognize 8 and 24 bit images\n");
return;
}

// Create and load the luminance scaling look up table.
float *luminance_lut = new float[256];
::load_luminance_lut(luminance_lut, m_pParams->GetGamma());

// Create and load the key look up table.
char *key_lut = new char[256];
::load_key_lut(key_lut, m_pParams->GetGain());

// Create and load the detail look up table.
float *detail_lut = new float[DETAILED_TOTAL];
const float lut_scale = (float)1.0; // Later this will be user controlled.
::load_detail_lut(detail_lut, m_pParams->GetLutScale());

// Determine which bit array to use for the reader's "crude metric"
// computation. If we have just signed this image, then use the
// true message bit array. Otherwise, we are trying to read
// without knowing the true message, and use the estimated
// message for computation of the metric.
unsigned char *referenceBitArray;
if (m_state == IMAGE_SIGNED || m_state == IMAGE_SIGNED_AND_VERIFIED ||
    m_state == IMAGE_SIGNED_AND_SAVED)
    referenceBitArray = m_pPackedMsg->getMsgBitArray();
else
    referenceBitArray = m_pPackedMsg->getReaderBitArray();

long data_length = signedImage.GetXDim() * signedImage.GetYDim();
long x_offset = 0;
long y_offset = 0;
int x_dim = signedImage.GetXDim();
int y_dim = signedImage.GetYDim();

if (signedImage.GetBitsPerPixel() == 8)
    num_channels = 1;
else if (signedImage.GetBitsPerPixel() == 24)
    num_channels = 3;

// See if we should use the super reader.
if (use_super_reader)
    reading_mode = 1;
else
    reading_mode = 0;

// Call the core recognizer
::read_8bit_single_channel_or_color(
    signedImage.GetPackedData(),
    x_dim,
    y_dim,
    x_offset,
    y_offset,
    x_dim,
    y_dim,
    // segment is full image.
    m_pPackedMsg->getMsgBitArrayLength(),
    signedImage.GetPackedData(),
    data_length,
    key_lut,
    luminance_lut,
    detail_lut,
    NULL,
    // No thumbnail at this time
    // unsignedImage.GetPackedData(),
    NULL, // Don't pass original data now
    (const unsigned char *) referenceBitArray,
    am_crude_metric,
    am_range,
    m_pPackedMsg->getReaderBitArray(),
    num_channels,
    reading_mode,
    m_pParams->GetBumpSize());

// Convert the recovered message bits back to an ASCII string.
m_pPackedMsg->bitToString();

TRACE ("The recognizer detected the following string: %s\n",
    m_pPackedMsg->getRecoveredAsciiMsg());

delete () luminance_lut;
delete () key_lut;
delete () detail_lut;
}

// CDibdoc commands

```

```

////////////////////
// OnSettingsSigner()
//
// This function is invoked when the user selects the Settings-->
// Signer Controls... menu item. It creates a Signer parameters
// dialog object and presents it to the user as a modal dialog.
// If the user presses OK, we then gather the new parameter values
// and use them to sign the image. Finally, a new view and window
// are created to display the signed image, if no such view already
// exists.
//
// CDibdoc::OnSettingsSigner()
{
    ParmDlg dlg;
    CRect rect;
    unsigned old_key;
    BOOL new_user_key = FALSE;

    // Check to see if we are in a legal state for signing.
    if (m_state == NO_IMAGE)
    {
        MessageBox(NULL,
            "An 8 or 24 bit image must be loaded before using the signer.",
            "Digital Signer Warning",
            MB_ICONINFORMATION | MB_OK);
        return;
    }

    // int scroll_pos

    // Initialize the dialog data
    dlg.m_message = m_pParams->GetMessage();
    dlg.m_gain_from_edit_box = m_pParams->GetGain();
    // dlg.m_gamma = m_pParams->GetGamma(); gamma no longer user cntrl
    dlg.m_key = m_pParams->GetKey();
    old_key = m_pParams->GetKey();
    dlg.m_bump_size = m_pParams->GetBumpSize();
    dlg.m_detail_lut_scale = m_pParams->GetLutScale();

    // Get the coordinates for the scroll bar object window.
    // dlg.m_gain.GetWindowRect(&rect);

    // Try to "create" the scroll bar.
    dlg.m_gain.Create(WS_CHILD, CRect(10, 50, 200, 20), &dlg, IDC_GAIN);

    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        // retrieve the dialog data
        m_pParams->SetMessage(dlg.m_message);

        if (dlg.m_key != old_key)
        {
            m_pParams->SetKey(dlg.m_key);
            new_user_key = TRUE;
        }

        m_pParams->SetGain(dlg.m_gain_from_edit_box);
        m_pParams->SetBumpSize(dlg.m_bump_size);
        m_pParams->SetLutScale(dlg.m_detail_lut_scale);
        // m_pParams->SetGamma(dlg.m_gamma); gamma no longer user cntrl

        // scroll_pos = dlg.m_gain.GetScrollPos();

        // TRACE("Scrollbar position: %d\n", scroll_pos);

        // This is going to take awhile
        BeginWaitCursor();

        // NOTE: AT THIS POINT SHOULD DETERMINE WHAT IMAGE IS IN THE
        // ACTIVE VIEW, AND IF IT CONTAINS A BITMAP SIGN THAT IMAGE.
        // SEE OnSettingsReader(), which uses the correct logic.
        // Then, call MakeSnow(himageToSignDib) and Sign(himageToSignDib)

        // If the user seed has changed, or if we haven't yet created
        // a coextensive key, create a snow image.
        if (new_user_key || m_hSnowyDIB == NULL)
            MakeSnow(m_hOriginalDIB);

        // Use the new settings, and sign the image.
        Sign();

        m_state = IMAGE_SIGNED;

        if (((CDiblookApp *)AfxGetApp())->m_autoread)

```

```

{
    // Run the reader again to see if we recover message.
    Read(m_IsignedDIB, FALSE);
}

m_state = IMAGE_SIGNED_AND_VERIFIED;

// Now see if a "signed image" view exists. If not, create it.
CreateUniqueView(SIGNED_VIEW);

// Now see if a "status image" view exists. If not, create it.
CvibView* p_statusview;
p_statusview = (CvibView *) CreateUniqueView(STATUS_VIEW);
EndWaitCursor();

// Refresh all of the views (Don't actually need to refresh Original one)
p_statusview->Doresize();
UpdateAllViews(NULL);

// Some debug stuff related to checksums.
TRACE("Signer checksum: %x\n", (int) m_pPackedMsg->GetSignerChecksum());
TRACE("Read checksum: %x\n", (int) m_pPackedMsg->GetReaderChecksum());
TRACE("Reader computed checksum: %x\n",
      (int) m_pPackedMsg->GetComputedReaderChecksum());
}

////////////////////////////////////
// CreateUniqueView()
////////////////////////////////////
// This function creates a new view of the indicated type, if and
// only if one does not already exist. It returns a pointer to
// the new view, if a new one is created, or a pointer to the
// pre-existing view of the specified type if one already exists.
// The "view_type" argument is one of the view types from SignView.h,
// i.e. SIGNED_VIEW, ORIGINAL_VIEW, STATUS_VIEW.
// CvibView* CreateUniqueView(int view_type)
// {
//     BOOL view_found = FALSE;
//     POSITION pos = GetFirstViewPosition();
//     CvibView* pview;
//     while (pos != NULL)
//     {
//         pview = GetNextView(pos);

//         // If we find it, we return the pointer and we're done.
//         if ( ((CvibView*)pview)->GetViewType() == view_type)
//             return pview;
//     }

//     // The desired type of view doesn't exist, so we create it.
//     CMainFrame* mMainFrame = (CMainFrame *) AfxGetApp()->m_pMainWnd;
//     mMainFrame->MyOnWindowNew();

//     // Now find the newly created view (last in list) and set its type.
//     pos = GetFirstViewPosition();
//     while (pos != NULL)
//     {
//         pview = GetNextView(pos);

//         ((CvibView*)pview)->SetViewType(view_type);
//         return(pview);
//     }

//     ChangeViewType()
//     //////////////////////////////////////
//     // This function finds the view of the "old type", and changes its
//     // type to "new type". If successful, it returns a pointer to
//     // the newly changed view. If not, returns NULL.
//     // The "view_type" arguments are from the view types in SignView.h,
//     // i.e. SIGNED_VIEW, ORIGINAL_VIEW, STATUS_VIEW, ALIGNED_VIEW.
//     CvibView* CDbiboc::ChangeViewType(int old_type, int new_type)
//     {
//         BOOL view_found = FALSE;
//         POSITION pos = GetFirstViewPosition();
//         CvibView* pview;
//         while (pos != NULL)
//         {
//             pview = GetNextView(pos);

//             // If we find it, change its type we return the pointer and we're done.
//             if ( ((CvibView*)pview)->GetViewType() == old_type)
//                 {

```

```

hImageToReadDIB = m_hOriginalDIB;
else if (view_type == SIGNED_VIEW)
hImageToReadDIB = m_hSignedDIB;
else if (view_type == ALIGNED_VIEW)
hImageToReadDIB = m_pAlignedImage->GetHIDIB();
else
{
    MessageBox(NULL, "Bug in OnSettingsReader!", "Error", MB_OK);
    return;
}

// Initialize the dialog data
dlg.m_user_key = m_pParams->GetKey();
old_key = m_pParams->GetKey();
dlg.m_msg_length = m_pParams->GetMessage().GetLength();
dlg.m_gain = m_pParams->GetGain();
dlg.m_bump_size = m_pParams->GetBumpSize();
dlg.m_detail_lut_scale = m_pParams->GetLutScale();
// dlg.m_use_super_reader = m_pParams->GetSuperReaderFlag();

// Invoke the dialog box
if (dlg.DoModal() == IDOK)
{
    m_pParams->SetGain(dlg.m_gain);
    m_pParams->SetBumpSize(dlg.m_bump_size);
    m_pParams->SetLutScale(dlg.m_detail_lut_scale);
    // m_pParams->SetSuperReaderFlag(dlg.m_use_super_reader);
    // If signer has not yet been used, or length changes, need a msg.
    if (m_pParams->GetMessage().GetLength() != (int) dlg.m_msg_length)
    {
        // Create a dummy msg of all x's.
        CString dummy_msg = CString('x', dlg.m_msg_length);
        m_pParams->SetMessage(dummy_msg);
    }

    // Create a PackedMsg object w/ our dummy msg.
    if (m_packedMsg != NULL)
        delete m_packedMsg;
    m_packedMsg = new PackedMsg( (const char *) m_pParams->GetMessage());
    if (dlg.m_user_key != old_key)
    {
        m_pParams->SetKey(dlg.m_user_key);
        new_user_key = TRUE;
    }

    // This is going to take awhile
    BeginWaitCursor();
    // If the user seed has changed, or if we haven't yet created
    // a coextensive key, create a snow image.
    if (new_user_key || m_hSnowDIB == NULL)
        MakeSnow(hImageToReadDIB);

    // Run the reader and attempt to recover message, and compute metrics.
    Read(hImageToReadDIB, m_pParams->GetSuperReaderFlag());

    // Make the state transition: depends on which image was read.
    if (view_type == ORIGINAL_VIEW || view_type == ALIGNED_VIEW)
        m_state = SUSPECT_READ;
    else if (view_type == SIGNED_VIEW)
    {
        if (m_state != IMAGE_SIGNED_AND_SAVED)
            m_state = IMAGE_SIGNED_AND_VERIFIED;
    }

    // KLUDGE for debug. Need the signer timestamp set.
    // WHY? 11/24
    m_pParams->UpdateSignTime();

    // Now see if a "status image" view exists. If not, create it.
    CDialog *p_statusView;
    p_statusView = (CDialog *) CreateUniqueView(STATUS_VIEW);
    EndWaitCursor();

    // Refresh all of the views (Don't actually need to refresh Original one)
    p_statusView->DoResize();
    UpdateAllViews(NULL);

    // See if the checksum read and the checksum computed from the
    // read message string agree. If not, warn user.
    if (m_packedMsg->GetReaderChecksum() !=
        m_packedMsg->GetComputedReaderChecksum())
    {
        MessageBox(NULL,
            "The embedded checksum didn't match the computed checksum.",
            "Warning", MB_OK);
    }
}

}

// Find the active view, determine its type, and return
// it to the caller. The type is one of those listed
// in the DIBView.h file.
int CDibDoc::GetActiveViewType(void)
{
    BOOL view_found = FALSE;
    POSITION pos = GetFirstViewPosition();
    CView* pView;
    while (pos != NULL)
    {
        pView = GetNextView(pos);

        // If we find it, we return the pointer and we're done.
        if ( ((CDibView*)pView)->IsActive() == TRUE)
            return ((CDibView*)pView)->GetViewType();
    }

    // We can get here when other apps are running and Windows sends message
    // resulting in CDibDoc::OnUpdateFileSaveAs() being called.
    // MessageBox(NULL, "Error in GetActiveViewType", "Error", MB_OK);
    return(UNKNOWN_VIEW);
}

// GetActiveView()
// Return a pointer to the active view (i.e., a CDibView*), or NULL
// if something goes wrong.
CDibView* CDibDoc::GetActiveView(void)
{
    BOOL view_found = FALSE;
    POSITION pos = GetFirstViewPosition();
    CView* pView;
    while (pos != NULL)
    {
        pView = GetNextView(pos);

        // If we find it, we return the pointer and we're done.
        if ( ((CDibView*)pView)->IsActive() == TRUE)
            return ((CDibView*)pView);
    }

    // We can get here when other apps are running and Windows sends message
    // resulting in CDibDoc::OnUpdateFileSaveAs() being called.
    // MessageBox(NULL, "Error in GetActiveViewType", "Error", MB_OK);
    return(NULL);
}

// OnSettingsAutoread()
// When the user toggles the "Auto-read after Signing" item in
// the Options menu, this function is invoked. It simply
// toggles the corresponding member variable.
// We currently also toggle the application level variable,
// so that the settings are global to all docs.
void CDibDoc::OnSettingsAutoread()
{
    if (m_autoread == TRUE)
    {
        m_autoread = FALSE;
        ((CDibDocApp *)AfxGetApp())->m_autoread = FALSE;
    }
    else
    {
        m_autoread = TRUE;
        ((CDibDocApp *)AfxGetApp())->m_autoread = TRUE;
    }
}

// OnUpdateSettingsAutoread()
// The framework calls this function whenever it is about
// to display the pulldown menu containing the Autoread
// option. Based on our internal state variable

```

```

// m_autoread, we set or clear the check mark next to
// the menu item using the pCmdUI->SetCheck() function.
// void CDibdoc::OnUpdateSettingsAutoread(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (((CDibLookApp *)AfxGetApp())->m_autoread == TRUE)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

// OnSettingsAlign()
// This function is called when the user selects the "Align" menu option.
// A CFileDialog object is created and used in order for the operator
// to specify the name of the "Reference Image" (a signed or unsigned
// original image used as the template).
// void CDibdoc::OnSettingsAlign()
{
    CString refname;
    BOOL success_flag;

    // Create a filter for the types of files the file dialog will offer
    char szFilter[] =
        "Windows Bit Map Files (*.bmp)|*.bmp|Device Independent Bitmaps (*.dib)|*.dib|*"
        "All Files (*.*)|*.*|*.*";

    // Construct a file dialog
    CFileDialog
        fileDlg(TRUE,
        "",
        NULL,
        OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        szFilter);

    // Over-ride the default title in the file dialog window
    fileDlg.m_ofn.lpstrTitle = "Select a template file to be used for alignment";

    // Display the file dialog
    if (fileDlg.DoModal() == IDOK)
    {
        // Get the name of the reference image file.
        refname = fileDlg.GetPathName();
        BeginWaitCursor();

        // Create an image object for the reference image.
        // If one already exists, delete it first.
        if (m_pRefImage != NULL)
            delete m_pRefImage;
        m_pRefImage = new Image(refname);

        if (m_pRefImage->GetFileOK == FALSE) // bail out if something went wrong
            return;

        // Display the reference image
        CreateUniqueView(REZ_VIEW);
        UpdateAllViews(NULL);

        TRACE("Call the Align() function (this is a test of trace output.)\n");
        // Do the actual alignment and change update the state description.
        success_flag = Align_it();
        if (success_flag)
        {
            m_state = SUSPECT_ALIGNED;

            // Now, the template image object has had its packed data array replaced
            // by the aligned, co-extensive image. Need to move this packed data
            // into the DIB array for display (and possible file saving) purposes.
            m_pRefImage->UnpackData();

            // We now call the image the Aligned image, not reference
            m_pAlignedImage = m_pRefImage;
            m_pRefImage = NULL;

            CreateUniqueView(ALIGNED_VIEW);

            // Create a status view, if it doesn't already exist.
            CDibView *p_statusView;
            p_statusView = (CDibView *) CreateUniqueView(STATUS_VIEW);
            p_statusView->DoResize();
            UpdateAllViews(NULL);
        }
    }
}

}

// EndWaitCursor();
}

// Align_it()
// This function is responsible for carrying out the alignment operation,
// by calling upon Geoff's core algorithms. It is assumed that on entry
// 1) m_nOriginalDIB is DIB of the suspect image, already loaded.
// 2) m_pRefImage points to a image object with the template (or
// reference) image.
// void CDibdoc::Align_it(void)
{
    int num_channels;

    // Create an image object for the suspect image.
    Image suspectImage(m_nOriginalDIB);

    // Currently we require that the reference and suspect are of same type
    // (i.e., both color or B&W).
    if (suspectImage.GetBitsPerPixel() != m_pRefImage->GetBitsPerPixel())
    {
        MessageBox(NULL,
        "The suspect and reference images must both be color or B&W",
        "Warning",
        MB_ICONINFORMATION | MB_OK);
        return(FALSE);
    }

    // Construct Align object.
    if (m_pAlign != NULL)
        delete m_pAlign;
    m_pAlign = new Align;

    // Create the "byte-wise" packed data arrays from the DIB 4-byte packing
    suspectImage.MakePackedData();
    m_pRefImage->MakePackedData();

    if (suspectImage.GetBitsPerPixel() == 8)
        num_channels = 1; // B&W image
    else if (suspectImage.GetBitsPerPixel() == 24)
        num_channels = 3; // Color image

    // Call the core algorithm to do the alignment.
    m_pAlign->direct_registration(m_pRefImage->GetPackedData(),
    m_pRefImage->GetKDIm(),
    m_pRefImage->GetKDIm(),
    suspectImage.GetPackedData(),
    suspectImage.GetKDIm(),
    suspectImage.GetKDIm(),
    num_channels);

    return(TRUE);
}

// OnUpdateFilesaveAs()
// When the file pulldown menu is selected, this function is called
// upon to determine whether the "Save As..." menu item should be
// enabled. It determines the type of the current view, and if it
// is of a type for which we currently allow file saves, the menu
// item is enabled.
// void CDibdoc::OnUpdateFilesaveAs(CCmdUI* pCmdUI)
{
    int view_type;

    // Determine the type of the current view.
    view_type = GetActiveViewType();

    // If the active view contains an image, we know how to save it.
    if (view_type == ORIGINAL_VIEW ||
    view_type == SIGNED_VIEW ||
    view_type == ALIGNED_VIEW ||
    view_type == STATUS_VIEW)
    {
        pCmdUI->Enable(TRUE);
    }
    else
}

```

```

}

pCmdui->Enable(FALSE);

}

//////////////////////////////////////////////////
// FILE: SignDoc.h
//////////////////////////////////////////////////
//
// DESCRIPTION:
// Interface file for the CDbDoc class. This defines the document class
// for the Signer. Under the Microsoft Foundation Class (MFC) architecture,
// the Document/View model is the preferred method. This header file
// defines our additions to the generic Document class created by the
// Visual C++ wizards.
//
// Copyright (C) 1996 Digimarc Corporation. All rights reserved.
//
//////////////////////////////////////////////////
//
#include "dibapi.h"
#include "packmsg.h"
#include "params.h"
#include "image.h"
#include "Align.h"
//
//
// Define the possible states...
#define NO_IMAGE 0
#define IMAGE_LOADED 1
#define IMAGE_SIGNED 2
#define IMAGE_SIGNED_AND_VERIFIED 3
#define SUSPECT_READ 4
#define IMAGE_SIGNED_AND_SAVED 5
#define SUSPECT_ALIGNED 6
//
// Include "signview.h"
//
// Define NO_IMAGE
// Define IMAGE_LOADED
// Define IMAGE_SIGNED
// Define IMAGE_SIGNED_AND_VERIFIED
// Define SUSPECT_READ
// Define IMAGE_SIGNED_AND_SAVED
// Define SUSPECT_ALIGNED
//
// For clarity when packing rgb images to 1 chan.
class CDbView;

class CDbDoc : public CDocument
{
protected: // create from serialization only
    CDbDoc();
    DECLARE_DYNCREATE(CDbDoc)

// Attributes
public:
    // HDIB GetHDIB() const
    // { return m_hDIB; }

    HDIB GetSignedHDIB() const
    { return m_hSignedDIB; }
    HDIB GetOriginalHDIB() const
    { return m_hOriginalDIB; }
    HDIB GetSnowyHDIB() const
    { return m_hSnowyDIB; }
    HDIB GetRefHDIB() const
    { return m_pRefImage->GetHDIB(); }
    HDIB GetAlignedHDIB() const
    { return m_pAlignedImage->GetHDIB(); }

    CPalette* GetDocPalette() const
    { return m_palDIB; }
    CSIZE GetDocSize() const
    { return m_sizeDoc; }

    PackedMsg* GetPackedMsg() const
    { return m_pPackedMsg; }

    SignerParams* GetSignerParams() const
    { return m_pParams; }

    int GetState() const { return m_state; }

    const CString& GetFilename() const { return m_filename; }

    float GetMetric() const { return m_crude_metric; }
    float GetRange() const { return m_range; }

    // Accessors so view objects can get alignment results.
    const AlignStatus GetAlignStatus(void) const { return m_pAlign->GetAlignStatus(); }
}

```


SIGNER. H


```

CPP_OBJS=.\Release\
CPP_SBRS=.\Release\
# ADD BASE MTL /nologo /D "NDEBUG" /win32
# ADD MTL /nologo /D "NDEBUG" /win32
MTL_PROJ=/nologo /D "NDEBUG" /win32
# ADD BASE RSC /I 0x409 /d "NDEBUG"
# ADD RSC /I 0x409 /d "NDEBUG"
RSC_PROJ=/I 0x409 /fo "$(INTDIR)/Signer.res" /d "NDEBUG"
BSC32-bcsmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o "$(OUTDIR)/SignerWin32.bsc"
BSC32_SBRS= \
    "$(INTDIR)/Mainfrm.sbr" \
    "$(INTDIR)/Sign.sbr" \
    "$(INTDIR)/Signdoc.sbr" \
    "$(INTDIR)/Coxkey.sbr" \
    "$(INTDIR)/Parmadlg.sbr" \
    "$(INTDIR)/Pft.sbr" \
    "$(INTDIR)/Stdafx.sbr" \
    "$(INTDIR)/Mychildw.sbr" \
    "$(INTDIR)/Packmsg.sbr" \
    "$(INTDIR)/Signview.sbr" \
    "$(INTDIR)/Myfile.sbr" \
    "$(INTDIR)/Params.sbr" \
    "$(INTDIR)/Signer.sbr" \
    "$(INTDIR)/Align.sbr" \
    "$(INTDIR)/Read.sbr" \
    "$(INTDIR)/Dibapi.sbr" \
    "$(INTDIR)/Readidl.sbr" \
    "$(OUTDIR)/SignerWin32.bsc" : "$(OUTDIR)" $(BSC32_SBRS)
$(BSC32) @<
$(BSC32_FLAGS) $(BSC32_SBRS)
<<
LINK32-link.exe
# ADD BASE LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /machine:IX86
# SUBTRACT LINK32 /profile /debug /nologo /subsystem:windows /machine:IX86
LINK32_FLAGS=oldnames.lib /nologo /stack:0x4000 /subsystem:windows \
/incremental:no /pdb:"$(OUTDIR)/SignerWin32.pdb" /machine:IX86 \
/def:"\Signer.def" /out:"$(OUTDIR)/SignerWin32.exe"
DEP_FILE=
"\Signer.def"
LINK32_OBJS= \
    "$(INTDIR)/Params.obj" \
    "$(INTDIR)/Signer.obj" \
    "$(INTDIR)/Align.obj" \
    "$(INTDIR)/Read.obj" \
    "$(INTDIR)/Dibapi.obj" \
    "$(INTDIR)/Readidl.obj" \
    "$(INTDIR)/Mainfrm.obj" \
    "$(INTDIR)/Sign.obj" \
    "$(INTDIR)/Signdoc.obj" \
    "$(INTDIR)/Coxkey.obj" \
    "$(INTDIR)/Parmadlg.obj" \
    "$(INTDIR)/Pft.obj" \
    "$(INTDIR)/Stdafx.obj" \
    "$(INTDIR)/Mychildw.obj" \
    "$(INTDIR)/Packmsg.obj" \
    "$(INTDIR)/Signview.obj" \
    "$(INTDIR)/Myfile.obj" \
    "$(INTDIR)/Image.obj" \
    "$(INTDIR)/Signer.res"
"$(OUTDIR)/SignerWin32.exe" : "$(OUTDIR)" $(LINK32_OBJS)
$(LINK32) @<
$(LINK32_FLAGS) $(LINK32_OBJS)
<<
ILBRIP "$(CFG)" == "Signer - Win32 Debug"
# PROP BASE Use_MFC 1
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir "Debug"
# PROP BASE Intermediate_Dir "Debug"
# PROP BASE Target_Dir ""
# PROP Use_MFC 1
# PROP Use_Debug_Libraries 1
# PROP Output_Dir "Debug"
# PROP Intermediate_Dir "Debug"
# PROP Target_Dir ""
OUTDIR=.\Debug
CLEAN :
ALL : "$(OUTDIR)\SignerWin32.exe" "$(OUTDIR)\SignerWin32.bsc"

```

```

-erase .\Debug\vc40.pdb"
-erase .\Debug\vc40.idb"
-erase .\Debug\SignerWin32.bsc"
-erase .\Debug\Dibapi.sbr"
-erase .\Debug\Readidl.sbr"
-erase .\Debug\Myfile.sbr"
-erase .\Debug\Mychildw.sbr"
-erase .\Debug\Coxkey.sbr"
-erase .\Debug\Signview.sbr"
-erase .\Debug\Signer.sbr"
-erase .\Debug\Stdafx.sbr"
-erase .\Debug\Read.sbr"
-erase .\Debug\Packmsg.sbr"
-erase .\Debug\Pft.sbr"
-erase .\Debug\Image.sbr"
-erase .\Debug\Parmadlg.sbr"
-erase .\Debug>Mainfrm.sbr"
-erase .\Debug\Signdoc.sbr"
-erase .\Debug\Align.sbr"
-erase .\Debug\Params.sbr"
-erase .\Debug\SignerWin32.exe"
-erase .\Debug\Params.obj"
-erase .\Debug\Dibapi.obj"
-erase .\Debug\Readidl.obj"
-erase .\Debug\Myfile.obj"
-erase .\Debug\Mychildw.obj"
-erase .\Debug\Coxkey.obj"
-erase .\Debug\Signview.obj"
-erase .\Debug\Signer.obj"
-erase .\Debug\Stdafx.obj"
-erase .\Debug\Read.obj"
-erase .\Debug\Packmsg.obj"
-erase .\Debug\Pft.obj"
-erase .\Debug\Sign.obj"
-erase .\Debug\Image.obj"
-erase .\Debug\Parmadlg.obj"
-erase .\Debug>Mainfrm.obj"
-erase .\Debug\Signdoc.obj"
-erase .\Debug\Align.obj"
-erase .\Debug\Signer.res"
*$(OUTDIR)" :
if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"
# ADD BASE CPP /nologo /MTd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D
# MBCS" /FR /YX /c
# ADD CPP /nologo /MTd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D "_MBCS" /FR
/YX /c
CPP_PROJ=/nologo /MTd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" \
/D "_MBCS" /FR "$(INTDIR)" /Fp "$(INTDIR)/SignerWin32.pch" /YX /Fo "$(INTDIR)" \
/Fd "$(INTDIR)" /c
CPP_OBJS=.\Debug\
CPP_SBRS=.\Debug\
# ADD BASE MTL /nologo /D "DEBUG" /win32
# ADD MTL /nologo /D "DEBUG" /win32
MTL_PROJ=/nologo /D "DEBUG" /win32
# ADD BASE RSC /I 0x409 /d "DEBUG"
# ADD RSC /I 0x409 /d "DEBUG"
RSC_PROJ=/I 0x409 /fo "$(INTDIR)/Signer.res" /d "DEBUG"
BSC32-bcsmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o "$(OUTDIR)/SignerWin32.bsc"
BSC32_SBRS= \
    "$(INTDIR)/Dibapi.sbr" \
    "$(INTDIR)/Readidl.sbr" \
    "$(INTDIR)/Myfile.sbr" \
    "$(INTDIR)/Mychildw.sbr" \
    "$(INTDIR)/Coxkey.sbr" \
    "$(INTDIR)/Signview.sbr" \
    "$(INTDIR)/Signer.sbr" \
    "$(INTDIR)/Stdafx.sbr" \
    "$(INTDIR)/Read.sbr" \
    "$(INTDIR)/Packmsg.sbr" \
    "$(INTDIR)/Pft.sbr" \
    "$(INTDIR)/Image.sbr" \
    "$(INTDIR)/Sign.sbr" \
    "$(INTDIR)/Parmadlg.sbr" \
    "$(INTDIR)/Mainfrm.sbr" \
    "$(INTDIR)/Signdoc.sbr" \
    "$(INTDIR)/Align.sbr" \
    "$(INTDIR)/Params.sbr" \
    "$(OUTDIR)/SignerWin32.bsc" : "$(OUTDIR)" $(BSC32_SBRS)
$(BSC32) @<
$(BSC32_FLAGS) $(BSC32_SBRS)
<<
LINK32-link.exe

```

```

# ADD BASE LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /debug /machine:IX86
# ADD LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /profile /debug /machine:IX86
LINK32_FLAGS=oldnames.lib /nologo /stack:0x2800 /subsystem:windows /profile /debug /machine:IX86
/bug_ /machine:IX86 /def:".\signer.def" /out:"$(OUTDIR)/SignerWin32.exe"
DEP_FILE= \
-.\signer.def"
LINK32_OBJS= \
$(INTDIR)/Params.obj" \
$(INTDIR)/Dibapi.obj" \
$(INTDIR)/ReadDlg.obj" \
$(INTDIR)/MyFile.obj" \
$(INTDIR)/MyChildw.obj" \
$(INTDIR)/Coxkey.obj" \
$(INTDIR)/Signview.obj" \
$(INTDIR)/Signer.obj" \
$(INTDIR)/Stdafx.obj" \
$(INTDIR)/Read.obj" \
$(INTDIR)/Packmsg.obj" \
$(INTDIR)/Fft.obj" \
$(INTDIR)/Image.obj" \
$(INTDIR)/Sign.obj" \
$(INTDIR)/Parmdlg.obj" \
$(INTDIR)/Mainfrm.obj" \
$(INTDIR)/Signdoc.obj" \
$(INTDIR)/Align.obj" \
$(INTDIR)/Signer.res"
"$(OUTDIR)\SignerWin32.exe" : "$(OUTDIR)" "$(DEP_FILE)" $(LINK32_OBJS)
$(LINK32) @<
$(LINK32_FLAGS) $(LINK32_OBJS)
<<
IENDIF

.c $(CPP_OBJS) \obj:
$(CPP) $(CPP_PROJ) <

.cpp $(CPP_OBJS) \obj:
$(CPP) $(CPP_PROJ) <

.cxx $(CPP_OBJS) \obj:
$(CPP) $(CPP_PROJ) <

.c $(CPP_SBRS) \abr:
$(CPP) $(CPP_PROJ) <

.cpp $(CPP_SBRS) \abr:
$(CPP) $(CPP_PROJ) <

.cxx $(CPP_SBRS) \abr:
$(CPP) $(CPP_PROJ) <

#####
# Begin Target
# Name "Signer - Win32 Release"
# Name "Signer - Win32 Debug"
IF "$(CFG)" == "Signer - Win32 Release"
ELSEIF "$(CFG)" == "Signer - Win32 Debug"
ENDIF
#####
# Begin Source File
SOURCE=.\Coxkey.cpp
DEP_CPP_COXKE= \
-.\Coxkey.h\
-.\Dibapi.h\
-.\Stdafx.h\
"$(INTDIR)\Coxkey.obj" : $(SOURCE) $(DEP_CPP_COXKE) "$(INTDIR)"
"$(INTDIR)\Coxkey.sbr" : $(SOURCE) $(DEP_CPP_COXKE) "$(INTDIR)"
# End Source File
#####
SOURCE=.\Dibapi.cpp
DEP_CPP_DIBAP= \
-.\Stdafx.h\
-.\Dibapi.h\
"$(INTDIR)\Dibapi.obj" : $(SOURCE) $(DEP_CPP_DIBAP) "$(INTDIR)"
DEP_CPP_PACKM= \
-.\Stdafx.h\
-.\Packmsg.h\
"$(INTDIR)\Packmsg.obj" : $(SOURCE) $(DEP_CPP_PACKM) "$(INTDIR)"
"$(INTDIR)\Packmsg.sbr" : $(SOURCE) $(DEP_CPP_PACKM) "$(INTDIR)"
# End Source File
#####
SOURCE=.\Fft.cpp
DEP_CPP_FFT= \
-.\Stdafx.h\
-.\Packmsg.h\
"$(INTDIR)\Fft.obj" : $(SOURCE) $(DEP_CPP_FFT) "$(INTDIR)"
"$(INTDIR)\Fft.sbr" : $(SOURCE) $(DEP_CPP_FFT) "$(INTDIR)"
# End Source File
#####
SOURCE=.\Image.cpp
DEP_CPP_IMAGE= \
-.\Image.h\
-.\Dibapi.h\
-.\Stdafx.h\
"$(INTDIR)\Image.obj" : $(SOURCE) $(DEP_CPP_IMAGE) "$(INTDIR)"
"$(INTDIR)\Image.sbr" : $(SOURCE) $(DEP_CPP_IMAGE) "$(INTDIR)"
# End Source File
#####
# Begin Source File
SOURCE=.\Mainfrm.cpp
IF "$(CFG)" == "Signer - Win32 Release"
DEP_CPP_MAINF= \
-.\Stdafx.h\
-.\Signer.h\
-.\Mainfrm.h\
-.\Params.h\
"$(INTDIR)\Mainfrm.obj" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"
"$(INTDIR)\Mainfrm.sbr" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"
ELSEIF "$(CFG)" == "Signer - Win32 Debug"
DEP_CPP_MAINF= \
-.\Stdafx.h\
-.\Signer.h\
-.\Mainfrm.h\
"$(INTDIR)\Mainfrm.obj" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"
"$(INTDIR)\Mainfrm.sbr" : $(SOURCE) $(DEP_CPP_MAINF) "$(INTDIR)"
ENDIF
#####
# End Source File
# Begin Source File
SOURCE=.\MyFile.cpp
DEP_CPP_MYFIL= \
-.\Stdafx.h\
-.\Dibapi.h\
"$(INTDIR)\MyFile.obj" : $(SOURCE) $(DEP_CPP_MYFIL) "$(INTDIR)"
"$(INTDIR)\MyFile.sbr" : $(SOURCE) $(DEP_CPP_MYFIL) "$(INTDIR)"
# End Source File
#####
SOURCE=.\Packmsg.cpp
DEP_CPP_PACKM= \
-.\Stdafx.h\
-.\Packmsg.h\
"$(INTDIR)\Packmsg.obj" : $(SOURCE) $(DEP_CPP_PACKM) "$(INTDIR)"
"$(INTDIR)\Packmsg.sbr" : $(SOURCE) $(DEP_CPP_PACKM) "$(INTDIR)"
# End Source File
#####
SOURCE=.\Params.cpp
DEP_CPP_PARAM= \
"$(INTDIR)\Params.obj" : $(SOURCE) $(DEP_CPP_PARAM) "$(INTDIR)"
"$(INTDIR)\Params.sbr" : $(SOURCE) $(DEP_CPP_PARAM) "$(INTDIR)"
# End Source File
#####
SOURCE=.\Signer.cpp
DEP_CPP_SIGNER= \
-.\Stdafx.h\
-.\Dibapi.h\
-.\Signer.h\
-.\Mainfrm.h\
-.\Params.h\
"$(INTDIR)\Signer.obj" : $(SOURCE) $(DEP_CPP_SIGNER) "$(INTDIR)"
"$(INTDIR)\Signer.sbr" : $(SOURCE) $(DEP_CPP_SIGNER) "$(INTDIR)"
# End Source File
#####
SOURCE=.\SignerWin32.exe
DEP_EXE_SIGNER= \
-.\Stdafx.h\
-.\Dibapi.h\
-.\Signer.h\
-.\Mainfrm.h\
-.\Params.h\
"$(INTDIR)\SignerWin32.obj" : $(SOURCE) $(DEP_EXE_SIGNER) "$(INTDIR)"
"$(INTDIR)\SignerWin32.sbr" : $(SOURCE) $(DEP_EXE_SIGNER) "$(INTDIR)"
# End Source File
#####

```



```

"$(INTDIR)\signdoc.obj" : $(SOURCE) $(DEP_CPP_SIGND) "$(INTDIR)"
"$(INTDIR)\signdoc.sbr" : $(SOURCE) $(DEP_CPP_SIGND) "$(INTDIR)"
!ENDIF
# End Source File
# Begin Source File
SOURCE=.\Signview.cpp
"..\stdafx.h"
"..\signer.h"
"..\signdoc.h"
"..\signview.h"
"..\dibapi.h"
"..\mainfrm.h"
"..\align.h"
"..\params.h"
"..\packmsg.h"
"..\image.h"
"$(INTDIR)\Signview.obj" : $(SOURCE) $(DEP_CPP_SIGNV) "$(INTDIR)"
"$(INTDIR)\Signview.sbr" : $(SOURCE) $(DEP_CPP_SIGNV) "$(INTDIR)"
# End Source File
# Begin Source File
SOURCE=.\Wychildw.cpp
!IF "$(CFG)" == "Signer - Win32 Release"
DEP_CPP_MYCHI=
"..\stdafx.h"
"..\signer.h"
"..\wychildw.h"
"..\Params.h"
"$(INTDIR)\Wychildw.obj" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"
"$(INTDIR)\Wychildw.sbr" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"
!ELSEIF "$(CFG)" == "Signer - Win32 Debug"
DEP_CPP_MYCHI=
"..\stdafx.h"
"..\signer.h"
"..\wychildw.h"
"$(INTDIR)\Wychildw.obj" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"
"$(INTDIR)\Wychildw.sbr" : $(SOURCE) $(DEP_CPP_MYCHI) "$(INTDIR)"
!ENDIF
# End Source File
# End Target
# End Project
#####
SIGNVIEW.CPP
#####
// Signview.cpp
// Implementation of the CDibView class
//
#include "stdafx.h"
#include "signer.h"
#include "signdoc.h"
#include "dibapi.h"
#include "mainfrm.h"
#include "Align.h"
#include <stretrea.h>
#include <omanip.h>
#ifdef _DEBUG
#define THIS_FILE
static char _BASED_CODE THIS_FILE[] = __FILE__
#endif
// CDibView
//
IMPLEMENT_DYNCREATE(CDibView, CScrollView)
BEGIN_MESSAGE_MAP(CDibView, CScrollView)
//{{AFX_MSG_MAP(CDibView)
ON_COMMAND(ID_EDIT_COPY, OnEditCopy)
ON_UPDATE_COMMAND_UI(ID_EDIT_COPY, OnUpdateEditCopy)
ON_COMMAND(ID_EDIT_PASTE, OnEditPaste)
ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE, OnUpdateEditPaste)
ON_MESSAGE(WM_DOREALIZE, OnDorealize)

```



```

ON COMMAND(ID_VIEW_SIGNED, OnViewSigned)
ON COMMAND(ID_VIEW_UNSIGNED, OnViewUnsigned)
ON COMMAND(ID_VIEW_SNOWY_IMAGE, OnViewSnowyImage)
ON COMMAND(ID_VIEW_STATUS, OnViewStatus)
ON UPDATE_COMMAND_UI(ID_VIEW_SIGNED, OnUpdateViewSigned)
ON UPDATE_COMMAND_UI(ID_VIEW_UNSIGNED, OnUpdateViewUnsigned)
ON UPDATE_COMMAND_UI(ID_VIEW_SNOWY_IMAGE, OnUpdateViewSnowyImage)
ON UPDATE_COMMAND_UI(ID_VIEW_STATUS, OnUpdateViewStatus)
//)AFX_MSG_MAP

// Standard printing commands
ON COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
ON COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

// CDbView()
// The constructor
//
// CDbView::CDbView()
//
// m_viewType = ORIGINAL_VIEW; // default type of view
// m_bThisViewActive = FALSE; // View is initially inactive
// m_bDoResizeStatusView = FALSE;
//
// - CDbView()
// The destructor.
//
// CDbView::~CDbView()
//
//
// Returns the HDIB (handle to the DIB) of the current view. Note that
// it doesn't make sense to call this if the current view is the status
// view, or any other view which isn't displaying a DIB.
//
// HDIB CDbView::GetHDIB(void)
// {
//     CDbDoc* pDoc = GetDocument();
//
//     switch (m_viewType)
//     {
//         case ORIGINAL_VIEW:
//             return pDoc->GetOriginalHDIB();
//             break;
//         case SIGNED_VIEW:
//             return pDoc->GetSignedHDIB();
//             break;
//         case SNOWY_VIEW:
//             return pDoc->GetSnowyHDIB();
//             break;
//         case RBF_VIEW:
//             return pDoc->GetRefHDIB();
//             break;
//         case ALIGNED_VIEW:
//             return pDoc->GetAlignedHDIB();
//             break;
//         case STATUS_VIEW:
//             return
//             default:
//             return pDoc->GetOriginalHDIB();
//             break;
//     }
// }

// OnDraw()
//
// Given a pointer to a CDC (device context), this function is responsible
// for drawing the current view.
//
// void CDbView::OnDraw(CDC* pDC)
// {
//     if (m_viewType == STATUS_VIEW)
//     {
//         DisplayStatus(pDC);
//     }
//     else
//     {
//         CDbDoc* pDoc = GetDocument();
//         HDIB hDIB = GetHDIB();

```

```

if (hDIB != NULL)
{
    LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);
    int cxDIB = (int) ::DIBWidth(lpDIB); // size of DIB - x
    int cyDIB = (int) ::DIBHeight(lpDIB); // size of DIB - y
    ::GlobalUnlock((HGLOBAL) hDIB);
    CRect rcDIB;
    rcDIB.top = rcDIB.left = 0;
    rcDIB.right = cxDIB;
    rcDIB.bottom = cyDIB;
    CRect rcDest;
    if (pDoc->IsPrinting()) // printer DC
    {
        // get size of printer page (in pixels)
        int cxPage = pDoc->GetDeviceCaps(HORZRES);
        int cyPage = pDoc->GetDeviceCaps(VERTRES);
        // get printer pixels per inch
        int cxInch = pDoc->GetDeviceCaps(LOGPIXELSX);
        int cyInch = pDoc->GetDeviceCaps(LOGPIXELSY);

        // Best Fit case -- create a rectangle which preserves
        // the DIB's aspect ratio, and fills the page horizontally.
        //
        // The formula in the "->bottom" field below calculates the y
        // position of the printed bitmap, based on the size of the
        // bitmap, the width of the page, and the relative size of
        // a printed pixel (cyinch / cxinch).
        //
        rcDest.top = rcDest.left = 0;
        rcDest.bottom = (int)((double)cyDIB * cxPage * cyinch)
            / ((double)cxDIB * cxInch);
        rcDest.right = cxPage;
    }
    else // not printer DC
    {
        rcDest = rcDIB;
    }
    ::PaintDIB(pDoc->m_hDC, rcDest, GetHDIB(), //pDoc->GetHDIB(),
        rcDIB, pDoc->GetDocPalette());
}

// OnPreparePrinting()
//
// BOOL CDbView::OnPreparePrinting(CPrintInfo* pInfo)
// {
//     // default preparation
//     return DoPreparePrinting(pInfo);
// }

// CDbView commands
//
// OnDIBRealize()
//
// LRESULT CDbView::OnDIBRealize(WPARAM wParam, LPARAM lParam)
// {
//     ASSERT(wParam != NULL);
//     CDbDoc* pDoc = GetDocument();
//     //if (pDoc->GetHDIB() == NULL)
//     if (GetHDIB() == NULL)
//         return 0L; // must be a new document
//
//     CPalette* pPal = pDoc->GetDocPalette();
//     if (pPal != NULL)
//     {
//         CMainFrame* pAppFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;
//         ASSERT(pAppFrame->IsKindOf(RUNTIME_CLASS(CMainFrame)));
//
//         CClientDC appDC(pAppFrame);
//         // All views but one should be a background palette.
//         // wParam contains a handle to the active view, so the SelectPalette
//         // beforeBackground flag is FALSE only if wParam == m_hWnd (this view)
//         CPalette* oldPalette = appDC.SelectPalette(pPal, (HWND)wParam) != m_hWnd;
//
//         if (oldPalette != NULL)
//         {
//             UINT nColorsChanged = appDC.RealizePalette();
//             if (nColorsChanged > 0)
//                 pDoc->UpdateAllViews(NULL);
//             appDC.SelectPalette(oldPalette, TRUE);
//         }
//         else

```

```

    {
        TRACE0("tselectPalette failed in CDibView::OnPaletteChanged\n");
    }
}

return 0L;

////////////////////////////////////
// OnInitialUpdate()
//
void CDibView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    ASSERT(GetDocument() != NULL);

    SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());
    // Resize this view's window based on the size of the image.
    ResizeParentToFit();

    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Original*");

    // OnActivateView()
    //
void CDibView::OnActivateView(BOOL bActivate, CView* pActivateView,
                             CView* pDeactivateView)
{
    CScrollView::OnActivateView(bActivate, pActivateView, pDeactivateView);
    if (bActivate)
    {
        m_bThisViewActive = TRUE;
        ASSERT(pActivateView == this);
        OnDoRealize((LPARAM)m_hWnd, 0); // same as SendMessage(WM_DOREALIZE);
    }
    else
    {
        m_bThisViewActive = FALSE;
    }

    // OnEditCopy()
    //
void CDibView::OnEditCopy()
{
    CDibDoc* pDoc = GetDocument();
    // Clean clipboard of contents, and copy the DIB.
    if (OpenClipboard())
    {
        BeginWaitCursor();
        EmptyClipboard();
        SetClipboardData(CF_DIB, CopyHandle((HANDLE) GetHDIIB())); //pDoc->GetHDIIB());
        CloseClipboard();
        EndWaitCursor();
    }
}

////////////////////////////////////
// OnUpdateEditCopy()
//
void CDibView::OnUpdateEditCopy(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(GetHDIIB() != NULL);
}

////////////////////////////////////
// OnEditPaste()
//
void CDibView::OnEditPaste()
{
    HDIB hNewDIB = NULL;
    if (OpenClipboard())
    {
        BeginWaitCursor();
        hNewDIB = (HDIB) CopyHandle(::GetClipboardData(CF_DIB));
        CloseClipboard();
        if (hNewDIB != NULL)

```

```

    {
        CDibDoc* pDoc = GetDocument();
        pDoc->ReplaceDIB(hNewDIB); // and free the old DIB
        pDoc->InitDIBData(); // set up new size & palette
        pDoc->SetModifiedFlag(TRUE);

        SetScrollSizes(MM_TEXT, pDoc->GetDocSize());
        OnDoRealize((LPARAM)m_hWnd, 0); // realize the new palette
        pDoc->UpdateAllViews(NULL);
    }
    EndWaitCursor();
}

////////////////////////////////////
// OnUpdateEditPaste()
//
void CDibView::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!IsClipboardFormatAvailable(CF_DIB));
}

////////////////////////////////////
// OnViewSigned()
//
void CDibView::OnViewSigned()
{
    CDibDoc* pDoc = GetDocument();
    m_viewType = SIGNED_VIEW;
    //pDoc->SetModifiedFlag(TRUE);

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Signed*");
    pDoc->UpdateAllViews(NULL);
}

////////////////////////////////////
// OnViewUnsigned()
//
void CDibView::OnViewUnsigned()
{
    CDibDoc* pDoc = GetDocument();
    m_viewType = ORIGINAL_VIEW;

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Original*");
    pDoc->UpdateAllViews(NULL);
}

////////////////////////////////////
// OnViewSnowyImage()
//
void CDibView::OnViewSnowyImage()
{
    CDibDoc* pDoc = GetDocument();
    m_viewType = SNOWY_VIEW;

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Code Pattern*");
    pDoc->UpdateAllViews(NULL);
}

////////////////////////////////////
// OnViewStatus()
//
void CDibView::OnViewStatus()
{
    CDibDoc* pDoc = GetDocument();
    m_viewType = STATUS_VIEW;

    // Set the window title.
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Status*");
    pDoc->UpdateAllViews(NULL);
}

```

```

////////////////////////////////////
// SetViewType()
////////////////////////////////////
void CDibView::SetViewType(int type)
{
    CDibDoc* pDoc = GetDocument();
    switch (type)
    {
        case SIGNED VIEW:
            m_viewType = SIGNED_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Signed*");
            break;

        case REP VIEW:
            m_viewType = REP_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Reference*");
            break;

        case ALIGNED VIEW:
            m_viewType = ALIGNED_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Aligned*");
            break;

        case STATUS VIEW:
            m_viewType = STATUS_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Status*");
            break;

        default:
            // This is an error.
            // atMessage
            break;
    }
}

```

```

////////////////////////////////////
// DisplayStatus()
////////////////////////////////////
void CDibView::DisplayStatus(CDC *pDC)
{
    CDibDoc* pDoc = GetDocument();
    TEXTMETRIC tm;
    CString text;
    CRect rect;
    Crime t;

    pDC->GetTextMetrics(&tm);

    int col = 20*tm.tmAveCharWidth;
    int line = tm.tmHeight;
    ostream strm;

    createStatusStream(strm);

    int height;
    rect.top = 10;
    rect.left = 10;
    rect.right = 50 * tm.tmAveCharWidth;

    height = pDC->DrawText(strm.str(), -1, &rect, DT_EXPANDTABS | DT_CALCRECT);
    rect.bottom = height + 10;
    pDC->DrawText(strm.str(), -1, &rect, DT_EXPANDTABS);

    // Resize the scrollbars to fit the information it contains.
    CSize size = CSize(rect.right+10, rect.bottom);
    SetScrollSizes(SM_TEXT, size);

    if (m_bDoResizeStatusView)
    {
        m_bDoResizeStatusView = FALSE;
        ResizeStatusView(size);
    }

    // Once we call .str(), we must delete the allocated space.
    delete strm.str();
    return;
}

```

```

////////////////////////////////////
// createStatusStream()
////////////////////////////////////
// Insert a stream of characters in to the ostream passed in by
// the caller, which describes the status. The state argument
// indicates our current program state, which influences what
// information is included in the stream data.
////////////////////////////////////
void CDibView::createStatusStream(ostream& strm)
{
    CDibDoc* pDoc = GetDocument();
    Crime t;
    int state = pDoc->GetState();
    PackedMsg* pMsg = pDoc->GetPackedMsg();

    strm << "\t\tSTATUS INFORMATION\n\n";

    switch (state)
    {
        case NO IMAGE:
            strm << "No image has been loaded.";
            break;

        case IMAGE LOADED:
            strm << "\tThe loaded image hasn't been signed or read.";
            break;

        case IMAGE SIGNED:
        case IMAGE SIGNED AND VERIFIED:
        case IMAGE SIGNED AND SAVED:
            strm << "Signer Status\n\n";
            strm << "\tOriginal Text:\t\t" << pMsg->getAsciiMsg() << "\n\n";
            strm << "\tMessage Length:\t\t" << pMsg->GetMsgLength() << "\n\n";
            strm << "\tGain Setting:\t\t" << pDoc->GetSignerParams()->GetGain() << "\n\n";
            // strm << "\tGamma:\t\t" << pDoc->GetSignerParams()->GetGamma() << "\n\n";
            strm << "\tKey:\t\t" << pDoc->GetSignerParams()->GetKey() << "\n\n";
            strm << "\tBump Size:\t\t" << pDoc->GetSignerParams()->GetBumpSize() << "\n\n";
            strm << "\tDetail Gain:\t\t" << pDoc->GetSignerParams()->GetDetailScale() << "\n\n";
            strm << "\tChecksum:\t\t" << (unsigned) pMsg->GetSignerChecksum() << "\n\n";

            strm.fill('0');
            t = pDoc->GetSignerParams()->GetTimestamp();
            strm << "\tTime of Signing:\t\t";

            // Disable the 4270 warning. This is a bug in Microsoft's fomanip.h.
            // without this, the setw() to manipulator causes a warning.
#pragma warning(disable:4270)
            strm << setw(2) << t.GetHour() << ':';
            strm << setw(2) << t.GetMinute() << ':';
            strm << setw(2) << t.GetSecond() << '.';
            strm << setw(2) << t.GetMonth() << '/';
            strm << setw(2) << t.GetDay() << '/';
            strm << setw(2) << t.GetYear() - 1900;
            strm << "\n\n";
            strm.fill(' ');

            // Put the warning level back to the default.
#pragma warning(default:4270)

            if (state == IMAGE_SIGNED_AND_SAVED)
                strm << "\tSigned image saved as:\t" << pDoc->GetFilename() << "\n\n";

            if (state == IMAGE_SIGNED_AND_VERIFIED)
            {
                strm << "Reader Status\n\n";
                strm << "\tRecognized Text:\t\t" << pMsg->getRecoveredAsciiMsg() << "\n\n";

                // Remove references to "super reader" for now
                //if (pDoc->GetSignerParams()->GetSuperReaderFlag())
                //    strm << "\tAlternative Reader:\t\t" << "On" << "\n\n";
                //else
                //    strm << "\tAlternative Reader:\t\t" << "Off" << "\n\n";

                // Adjust the floating point precision of the stream.
                strm.setf(ios::fixed, ios::floatfield);
                strm.precision(2);

                strm << "\tBit Success Rate (%) \t\t" << pMsg->GetPercentCorrect() << "\n\n";
            }
    }
}

```

```

    // Print crude metric.
    strm.precision(4);
    strm << "\tBit Estimator Std. Dev.: \t" << pDoc->GetMetric() << "\n\n";

    // Print range.
    strm << "\tBit Estimator Range: \t" << pDoc->GetRange() << "\n\n";

    strm << "\tEmbedded Checksum Read: \t" << (unsigned) pMsg->GetReaderChecksum()
    << "\n\n";

    strm << "\tChecksum Calculated: \t" << (unsigned) pMsg->GetComputedReaderChecksum()
    << "\n\n";
}

break;

case SUSPECT_ALIGNED:
    AlignStatus a_stats = pDoc->GetAlignStatus(); // Get the align status
    strm << "Aligned Image Status\n\n";

    // Adjust the floating point precision of the stream.
    strm.setf(ios::fixed, ios::floatfield);
    strm.precision(2);

    strm << "\tRotation applied to suspect: \t" << a_stats.rotation << "\n\n";
    strm << "\tTranslation (X, Y): \t" << a_stats.x_trans
    << ", " << a_stats.y_trans << "\n\n";
    strm << "\tScaling (X, Y): \t" << a_stats.x_scale
    << ", " << a_stats.y_scale << "\n\n";
    strm << "\tRefinement: \t" << a_stats.refinement << "\n\n";

    break;

case SUSPECT_READ:
    strm << "Reader Status\n\n";

    strm << "\tAssumed Message Length: \t" << pMsg->GetMsgLength() << "\n\n";

    strm << "\tRecognized Text: \t" << pMsg->GetRecoveredAsciiMsg() << "\n\n";

    strm << "\tAssumed Key: \t" << pDoc->GetSignerParams()->GetKey() << "\n\n";

    strm << "\tBump Size: \t" << pDoc->GetSignerParams()->GetBumpSize() << "\n\n";

    strm << "\tDetail Gain: \t" << pDoc->GetSignerParams()->GetDetailScale() << "\n\n";

    // Remove references to "super reader" for now
    //if (pDoc->GetSignerParams()->GetSuperReaderFlag())
    //    strm << "\tAlternative Reader: \t" << "On" << "\n\n";
    //else
    //    strm << "\tAlternative Reader: \t" << "Off" << "\n\n";

    // Adjust the floating point precision of the stream.
    strm.setf(ios::fixed, ios::floatfield);
    strm.precision(2);

    // Print crude metric.
    strm.precision(4);
    strm << "\tBit Estimator Std. Dev.: \t" << pDoc->GetMetric() << "\n\n";

    // Print range.
    strm << "\tBit Estimator Range: \t" << pDoc->GetRange() << "\n\n";

    strm << "\tEmbedded Checksum Read: \t" << (unsigned) pMsg->GetReaderChecksum()
    << "\n\n";

    strm << "\tChecksum Calculated: \t" << (unsigned) pMsg->GetComputedReaderChecksum()
    << "\n\n";

    break;
default:
    break;
}

// Add a null terminator (DrawText needs it).
strm << '\0';

// ResizeStatusView()
//
// Resizes the status view frame window. The goal is to not
// move the upper left corner, and to not exceed the bounds of
// the MDI main frame window on the right or left borders.
void CDibView::ResizeStatusView(CSize status_size)
{
    const int bar_height = 27; // An empirically derived kludge

```

SIGNVIEW.H

```

// signview.h : interface of the CDbView class
//
#include <strstream.h>

// Here I define the different types of views.
#define UNKNOWN_VIEW -1
#define SIGNED_VIEW 1
#define ORIGINAL_VIEW 2
#define SNOWY_VIEW 3
#define STATUS_VIEW 4
#define REF_VIEW 5
#define ALIGNED_VIEW 6

// reference image for alignment
// image after alignment completed

class CDbView : public CScrollView
{
public:
    CDbView();
    DECLARE_DYNCREATES(CDbView)

    // Attributes
public:
    CDbDoc* GetDocument()
    {
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDbDoc)));
        return (CDbDoc*) m_pDocument;
    }

private:
    int m_viewType;
    BOOL m_bThisViewActive;
    BOOL m_bResizeStatusView;

    // Operations
public:
    // Implementation
public:
    virtual ~CDbView();
    virtual void OnDraw(CDC* pDC); // overridden to draw this view

    virtual void OnInitialUpdate();
    virtual void OnActivateView(BOOL bActivate, CView* pActivateView,
        CView* pDeactivateView);
    void SetViewType(int type);
    int GetViewType(void) { return m_viewType; }
    BOOL IsViewActive(void) { return m_bThisViewActive; }
    void ResizeStatusView(BOOL bResizeStatusView = TRUE);
    void ResizeStatusView(CSize status_size);

    // I need OnFilePrint to be accessible from outside.
    void OnFilePrint(void) { CScrollView::OnFilePrint(); }

    void CreateStatusStream(ostrstream &strm);

    // Printing support
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);

private:
    HDIB GetDIB(void);
    void CDbView::DisplayStatus(CDC *pDC);

    // Generated message map functions
protected:
    ///({AFX_MSG(CDbView)
    afx_msg void OnEditCopy();
    afx_msg void OnUpdateEditCopy(CCmdUI* pCmdUI);
    afx_msg void OnEditPaste();
    afx_msg void OnUpdatePaste(CCmdUI* pCmdUI);
    afx_msg LRESULT OnDoRealize(WPARAM wParam, LPARAM lParam); // user message
    afx_msg void OnViewSigned();
    afx_msg void OnViewUnsigned();
    afx_msg void OnViewSnowyImage();
    afx_msg void OnViewStatus();
    afx_msg void OnUpdateViewSigned(CCmdUI* pCmdUI);
    afx_msg void OnUpdateViewSnowyImage(CCmdUI* pCmdUI);
    afx_msg void OnUpdateViewStatus(CCmdUI* pCmdUI);
    afx_msg void OnUpdateViewUnsigned(CCmdUI* pCmdUI);
    ///})AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

SNOWTMP.CPP

```

////////////////////
// My experimental member function which
// builds a snowy image in place.
//
////////////////////
void CDbDoc::MakeSnow(void)
{
    int cxDIB, cyDIB;
    long num_pixels, num_colors;
    LPSTR lpDIB, lpSnowyDIB;
    LPBITMAPINFOHEADER lpDIBHdr, lpSnowyDIBHdr;
    LPSTR lpDIBBits; // Pointer to DIB bits
    char __huge *src_data, *dest_data; // Huge ptrs for copying the image.

    HDIB hUnsignedDIB = GetHDIB();
    if (hUnsignedDIB == NULL)
        return;

    // Create space for the unsigned DIB for the snowy image.
    m_hSnowyDIB = (HDIB)::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, m_dwTotalDIBSize);
    if (m_hSnowyDIB == 0)
        return;

    // Here I follow the similar code in PaintDIB() of dibapi.cpp
    lpDIB = (LPSTR)::GlobalLock((HGLOBAL) hUnsignedDIB);
    lpSnowyDIB = (LPSTR)::GlobalLock((HGLOBAL) m_hSnowyDIB);
    src_data = (char __huge *) lpDIB;
    dest_data = (char __huge *) lpSnowyDIB;

    // Copy the BITMAPINFOHEADER, palette, and actual image byte data.
    for (image_byte = 0; image_byte < m_dwTotalDIBSize; image_byte++)
    {
        dest_data++ = src_data++;
    }

    lpDIBHdr = (LPBITMAPINFOHEADER) lpDIB; // Ptr to bitmap info hdr at start of dib.

    // Get ptr to the snowy dib header space, and copy header into it.
    lpSnowyDIBHdr = (LPBITMAPINFOHEADER) lpSnowyDIB;
    *lpSnowyDIBHdr = *lpDIBHdr;

    lpDIBBits = ::FindDIBBits(lpDIB);
    lpSnowyDIBBits = ::FindDIBBits(lpSnowyDIB);
    src_data = (char __huge *) lpDIBBits;
    dest_data = (char __huge *) lpSnowyDIBBits;

    // Copy the actual image byte data.
    for (image_byte = 0; image_byte < m_dwTotalDIBSize; image_byte++)
    {
        dest_data++ = src_data++;
    }

    cxDIB = (int) ::DIBWidth(lpDIB); // X size of DIB
    cyDIB = (int) ::DIBHeight(lpDIB); // Y size of DIB
    num_pixels = (long) cxDIB * cyDIB;
    num_colors = ::DIBNumColors(lpDIB);
    if (lpDIBHdr->biCompression != 0)
    {
        TRACE("Can't cope with compressed image (compression = %d)\n",
            ::GlobalUnlock((HGLOBAL) hUnsignedDIB);
        return;
    }
    lpDIBHdr->biCompression = 0;

    TRACE("width = %d, height = %d, num_pixels = %ld\n", cxDIB, cyDIB, num_pixels);
    TRACE("num_colors = %d\n", num_colors);
    if (num_colors == 0 || num_colors == 16)

```

```

TRACE("At this time, only build snowy image for 8 bit images\n");
::GlobalUnlock((HGLOBAL) hUnsignedDIB);
return;
}

if (num_colors == 256)
{
    Coxkey coxkey(1, (BITMAPINFO *) lpDIBHdr, lpDIBBits);
}

::GlobalUnlock((HGLOBAL) hUnsignedDIB);
}

// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.

// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

// stdafx.cpp : source file that includes just the standard includes
// stdafx.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information
#include "stdafx.h"

// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992 Microsoft Corporation
// All rights reserved.

// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
#include <afxwin.h> // MFC core and standard components

// SIGN PUBLIC.CPP
// FILE: Sign_Public.cpp
// DESCRIPTION:
// Core signing functions of the public digimarc technology.
// Started late April 1996
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.

#include "sign.h"
#include <math.h>
#include "stdafx.h"

#define SIGNATURE_BLOCK_DIMENSION 128
#define HIGHEST_GREY_VALUE 255
#define GRID_MINIMUM_GAIN -0.5
#define RED_DOG 0.33
#define GREEN_DOG 0.34
#define BLUE_DOG 0.33

// this function simply loads the floating point values of the bumps for a given "bump raster line"
// the output of this function (the bump array) should be roughly similar no matter
// what the bump size is or whether you're dealing with color or B&W
// REMEMBER: this function pads the ends on each side with one extra bump
int load_bump_array(
    float *bump, // floating point bump array to be filled (output)
    unsigned char *data, // input pixel data
    long xdim, // number of bumps in this row (not pixels), add 2 for output
    long zdim, // number of channels
    long bump_size, // pixels per bump
    long jump_x, // number of raw pixels between (xdim*bump_size) and entire image array x
    dimension
)
{
    long overfill; // this tells the innards that the incoming bump array needs a copied value
    into the first and last place
    {
        unsigned char *pdata;
        long i,j,k;
        float *pbump, bump_squared = (float)bump_size * (float)bump_size;

        pdata = data;
        if(overfill)pbump = bump+1;
        else pbump = bump;
        if(zdim == 1) { // single channel
            if(bump_size == 1) {
                for(j=0;j<xdim;j++)*(pbump++) = (float) * (pdata++);
            }
            else if(bump_size == 2) {
                // zero out bump array
                memset(bump,0,(xdim+2)*sizeof(float));
                for(i=0;i<2;i++) {
                    if(overfill)pbump = bump+1;
                    else pbump = bump;
                    for(j=0;j<xdim;j++) {
                        *pbump++ = (float) * (pdata++);
                        *pbump++ += (float) * (pdata++);
                    }
                    pdata += jump_x;
                }
                if(overfill)pbump = bump+1;
                else pbump = bump;
                for(i=0;i<xdim;i++)*(pbump++) /= bump_squared;
            }
            else {
                // zero out bump array
                memset(bump,0,(xdim+2)*sizeof(float));
                for(i=0;i<bump_size;i++) {
                    if(overfill)pbump = bump+1;
                    else pbump = bump;
                    for(j=0;j<xdim;j++) {
                        *pbump++ = bump;
                        for(k=0;k<bump_size;k++)*pbump++*(pdata++);
                        pbump++;
                    }
                    pdata += jump_x;
                }
                if(overfill)pbump = bump+1;
                else pbump = bump;
                for(i=0;i<xdim;i++)*(pbump++) /= bump_squared;
            }
        }
        else { // multi-channel, ASSUME ONLY RGB and GREEN channels at present
            float red = (float)RED_DOG, green=(float)GREEN_DOG, blue=(float)BLUE_DOG;
            if(bump_size == 1) { // this case is split off only for a X4 speed increase in
                execution
                for(j=0;j<xdim;j++) {
                    *pbump = red * (float) * (pdata++); // gimme an R
                    *pbump += green * (float) * (pdata++); // gimme a G
                    *pbump++ += blue * (float) * (pdata++); // gimme a B
                }
            }
            else {
                // zero out bump array
                memset(bump,0,(xdim+2)*xdim*sizeof(float));
                for(i=0;i<bump_size;i++) {
                    if(overfill)pbump = bump+1;
                    else pbump = bump;
                    for(j=0;j<xdim;j++) {
                        *pbump++ = bump;
                        for(k=0;k<bump_size;k++) {
                            *pbump++ = red * (float) * (pdata++); // gimme an R
                            *pbump++ = green * (float) * (pdata++); // gimme a G
                            *pbump++ = blue * (float) * (pdata++); // gimme a B
                        }
                        pbump++;
                    }
                    pdata += zdim * jump_x;
                }
                if(overfill)pbump = bump+1;
                else pbump = bump;
                for(i=0;i<xdim;i++)*(pbump++) /= bump_squared;
            }
        }
        // fill the end two values
        if(overfill) {
            bump[0]-bump[1];
            bump[xdim+1]-bump[xdim];
        }
    }
}

```

```

    return(1);
}

// load_funky_lut()
// This function loads the scaling factor based on minimum linear funkiness
int load_funky_lut( float *funky_lut ) // explicitly written for 8 bit
{
    int i,status=1,detail_start,detail_stop;
    float length;

    float scale = (float)1.0;
    detail_start = 1;
    detail_stop = 50;
    length = (float)detail_stop - (float)detail_start;
    for(i=0;i<detail_start;i++)funky_lut[i]=(float)0.0;
    for(i=detail_start;i<detail_stop;i++)
    {
        funky_lut[i] = scale*((float)(i-detail_start)/length);
    }
    for(i=detail_stop;i<512;i++)funky_lut[i]=funky_lut[detail_stop-1];
    return(status);
}

// this function associates a given row and column value of a bump in the
// standard signature block with A) the bit plane of the message associated with the bump,
// output in the message_bit_lut variable array, and B) whether the '1' direction is up
// XOR lut=1 down, XOR lut=0 up
// IMPORTANT: this also takes care of the basic XOR'ing operation between the message and
// the underlying code pattern (invert, don't invert)
int load_standard_message_block_lut(
    unsigned char *message, // if this is NULL, return the un XOR'ed array (for reading)
    long message_length,
    unsigned char *control_message, // this is the separate "always gotta be there" message
    long control_message_length, // its length
    short *message_bit_lut,
    unsigned char *XOR_lut,
    long read_or_write
){
    // this is a crude first version... April 1996
    // we're going with 16 control bits, and in this demo, we'll use all of them
    // to describe the raw message length as a short unsigned int

    //int length_table = new int(15);
    //int *xblocks = new int(15);
    //int *yblocks = new int(15);
    int length_table[] = {16,24,32,48,64,96,128,192,256,384,512,768,1024,1536,3072};
    int xblocks[] = { 8, 8, 4, 8, 4, 2, 4, 2, 4, 2, 1, 2, 1, 2, 1 };
    int yblocks[] = { 8, 8, 8, 8, 4, 8, 4, 2, 4, 2, 2, 1, 2, 1, 1 };

    // find which length in the length table is next highest over current message_length
    long index=0;
    while( length_table[index] < message_length ){
        index++;
    }

    long length = (SIGNATURE_BLOCK_DIMENSION/2)/xblocks[index]; // length in bumps
    long ylength = (SIGNATURE_BLOCK_DIMENSION/2)/yblocks[index];
    long current_bit=kfoo.lfoo;
    long jump = SIGNATURE_BLOCK_DIMENSION;
    short actual_bit;
    long one;
    long i,j,k,l;
    short *pmessage_bit;
    unsigned char *pxor;
    for(i=0;i<yblocks[index];i++){
        current_bit = 1*i+j*length; // this is
        // simply a "mixing agent" so that given bit planes
        // don't congregate around edges (come up with a better way please please
        // the following uses the
        // 1 0
        // 0 1
        // formula of local bumps associated with a given bit plane, hence the 2's
        // floating around
        for(k=0;k<ylength;k++){
            // reset the pointers
            pmessage_bit = &message_bit_lut[2*j*length + 2*(i*length+k)*jump];

```

```

        pdata += jump_x;
        pdata_out += jump_x;
    }
}

else { // multi-channel, assume ONLY RGB and three channels at present
    float red = (float)RED_DOG, green=(float)GREEN_DOG, blue=(float)BLUE_DOG;
    float red_ratio, green_ratio, blue_ratio, lum_zero = (float)0.1;
    if (bump_size == 1) {
        for (j=0; j<xdim; j++) {
            lum = red * (float)pdata + green * (float)(pdata+1) + blue * (float)(pdata+2);
            if (lum==zero) {
                red_ratio = (float)(pdata++) / lum;
                green_ratio = (float)(pdata++) / lum;
                blue_ratio = (float)(pdata++) / lum;
            }
            else {
                red_ratio = green_ratio = blue_ratio = (float)1.0;
                pdata+=3;
            }
            lum += *(ptweak++);
            // red
            temp = (int)( lum * red_ratio + half );
            if (temp<0) *(pdata_out++)=0;
            else if (temp>HIGHEST_GREY_VALUE) *(pdata_out++)=(unsigned char)HIGHEST_GREY_VALUE;
            else *(pdata_out++) = (unsigned char)temp;
            // green
            temp = (int)( lum * green_ratio + half );
            if (temp<0) *(pdata_out++)=0;
            else if (temp>HIGHEST_GREY_VALUE) *(pdata_out++)=(unsigned char)HIGHEST_GREY_VALUE;
            else *(pdata_out++) = (unsigned char)temp;
            // blue
            temp = (int)( lum * blue_ratio + half );
            if (temp<0) *(pdata_out++)=0;
            else if (temp>HIGHEST_GREY_VALUE) *(pdata_out++)=(unsigned char)HIGHEST_GREY_VALUE;
            else *(pdata_out++) = (unsigned char)temp;
        }
    }
    else {
        for (i=0; i<bump_size; i++) {
            ptweak = tweak;
            for (j=0; j<xdim; j++) {
                for (k=0; k<bump_size; k++) {
                    lum = red * (float)pdata + green * (float)(pdata+1) + blue *
                        (float)(pdata+2);
                    if (lum==zero) {
                        red_ratio = (float)(pdata++) / lum;
                        green_ratio = (float)(pdata++) / lum;
                        blue_ratio = (float)(pdata++) / lum;
                    }
                    else {
                        red_ratio = green_ratio = blue_ratio = (float)1.0;
                        pdata+=3;
                    }
                    lum += *ptweak;
                    // red
                    temp = (int)( lum * red_ratio + half );
                    if (temp<0) *(pdata_out++)=0;
                    else if (temp>HIGHEST_GREY_VALUE) *(pdata_out++)=HIGHEST_GREY_VALUE;
                    else *(pdata_out++) = (unsigned char)temp;
                    // green
                    temp = (int)( lum * green_ratio + half );
                    if (temp<0) *(pdata_out++)=0;
                    else if (temp>HIGHEST_GREY_VALUE) *(pdata_out++)=HIGHEST_GREY_VALUE;
                    else *(pdata_out++) = (unsigned char)temp;
                    // blue
                    temp = (int)( lum * blue_ratio + half );
                    if (temp<0) *(pdata_out++)=0;
                    else if (temp>HIGHEST_GREY_VALUE) *(pdata_out++)=HIGHEST_GREY_VALUE;
                    else *(pdata_out++) = (unsigned char)temp;
                    ptweak++;
                }
            }
            pdata += jump_x*xdim;
            pdata_out += jump_x*xdim;
        }
    }
    return (1);
}

// core_sign_public_generation_1()
//
// problem has been reduced to basic block unit;
// the only special case is when xdim and/or ydim are not extended to full block size
//
int core_sign_public_generation1()

```

```

    unsigned char *data, // pointer to upper left corner of image block
    long xdim, // absolute pixel dimension of current block
    long ydim, // absolute pixel dimension of entire original image or passed array
    long xdim, // absolute pixel dimension of current block
    long bump_size, // number of channels, e.g. 3 for RGB
    long message_length, // message length
    short *message_bit_lut,
    unsigned char *xor_lut, // this can be economized and reduced by 8 by using bitwise
    packing (I don't bother here)
    float *luminance_lut,
    float *detail_lut,
    float *subliminal_grid,
    unsigned char *pdata_out, // NULL if data is to be put back into input array
    float global_gain,
    float asymmetric_gain,
    float *funky_lut
) {
    long jump_x = Original_xdim - xdim; // this is the pointer offset for jumping rows
    unsigned char *pdata_out;
    long i, j;
    float *p1, *p2, *p3, *p4, *pbump, local_average_gain, detail_gain, diff;
    float *psubliminal_grid, lum_gain, asym_gain, funky_gain;
    short *pb1;
    unsigned char *pxor;
    double dtimp, bottomfunk;

    // set pdata_out based on (in place) versus new output array
    if (data_out == NULL) pdata_out = data;
    else pdata_out = data_out;

    // calculate bitwise bias between original image, (optionally degraded by common-model
    // distortion), and each bit of the message; this will be used for differential gain of
    // the bit planes to help "struggling" bits
    float *bit_bias = new float[message_length];
    for (i=0; i<message_length; i++) bit_bias[i] = (float)1.0;
    // read block signature
    // convert_read_to_bias
    //
    // dive into main loop
    //
    Main loop version 1 works in the following way. It is designed so that it can
    create a lagged version of the output in order to support either case of: A) where
    the input data array is replaced with the output array (in place), or B) where the
    *data_out pointer is not null and is the actual output array.
    _____ THIS PARTICULAR VERSION EXPECTS case B _____

    The main loop essentially operates bump by bump. It determines the local overall
    gain that should be applied to the given bump, then tweaks the individual pixel(s)
    of the output bump and stores in the temporary array which is later written out into
    the ultimate output array.
    *
    //
    long xbumpdim = xdim/bump_size; // calling routine guaranteed this would never have a
    remainder
    long ybumpdim = ydim/bump_size;
    // create initial bump arrays
    int xbumpsize = xbumpdim*2; // adding '2' allows us to not worry about edges in core loops
    float *bump0 = new float[xbumpsize];
    float *bump1 = new float[xbumpsize];
    float *bump2 = new float[xbumpsize];
    // load row 1 and row 2 (with row 0 data) for the first process step
    //
    // load bump array should copy elements 0 and 1 with data bump 0
    // and elements xbumpdim and xbumpdim+1 with data bump xbumpdim-1
    load_bump_array(bump1, data, xbumpdim, xdim, bump_size, jump_x, 1);
    memcpy(bump2, bump1, xbumpsize*sizeof(float));
    // create tweak array for each raster of bumps
    float *ptweak = new float[xbumpdim];
    float *tweak;
    float f1 = (float)1.0;
    float f4 = (float)4.0;
    for (i=0; i<ybumpdim; i++) {
        // in order to avoid modulo housekeeping later on, copy the arrays downward
        // (as they are small too)
        memcpy(bump0, bump1, xbumpsize*sizeof(float));
        memcpy(bump1, bump2, xbumpsize*sizeof(float));
        if (i != (ybumpdim-1)) { // load next bump row array
            load_bump_array(bump2, data, (i+1)*bump_size-Original_xdim, xbumpdim, xdim, bump_size, jump_x,
                1);
        }
        else { // leave bump2 alone
            p1 = bump0+1;
            p2 = bump1;
            p3 = bump2+1;
            p4 = bump1+2;
            pbump = bump1+1;
            psubliminal_grid = subliminal_grid + subliminal_grid*(SIGNATURE_BLOCK_DIMENSION);
            ptweak = tweak;
        }
    }
}

```



```

pbit = smessage bit_lut[!(SIGNATURE BLOCK DIMENSION)];
pxor = xor_lut[!(SIGNATURE BLOCK DIMENSION)];
for(j=0;j<xbmpdim;j++){ // this is the heart of the signing code and process, one bump at a
time

```

```

/* Here's the deal: (Written 4/26/96)
The goal of the signing process, beyond simply functioning,
is to maximize the "numeric detectability" of an embedded signature
while meeting some form of fixed "visibility/acceptability threshold" set by
a given user/creator.

```

In service to design toward this goal, imagine the following three axis parameter space, where two of the axes are only half-axis (positive only), and the third is a full axis (both negative and positive). This set of axes define two of the usual eight octal spaces of euclidean 3-space. As things define and "deservably separable" parameters show up on the scene (such as "extended local visibility metrics"), then they can define their own (generally) half-axis and extend the following example beyond three dimensions.

The signing design goal becomes optimally assigning a "gain" to a local bump based on its coordinates in the above defined space, whilst keeping in mind the basic needs of doing the operations fast in real applications. To begin with, the three axes are the following. We'll call the two half axes x and y, while the full axis will be z.

The x axis represents the luminance of the singular bump. The basic idea is that you can squeeze a little more energy into bright regions, as opposed to dim ones. It is important to note that when true "psycho-linear" device independent luminance values (pixel DN's) come along, this axis might become superfluous, unless of course if the luminance value couples into the other operative axes (e.g. C*xy). For now, this is here as much due to the sub-optimality of current quasi-linear luminance coding.

The y axis is the kitchen sink of "local hiding potential" of the neighborhood within which the bump finds itself. The basic idea is that flat regions have a low hiding potential since the eye can detect subtle changes in such regions, whereas complex textured regions have a high hiding potential. Long lines and long edges tend toward the lower hiding potential, since "breaks and chopiness" in nice smooth long lines are somewhat visible, while shorter lines and edges, and mosaics thereof tend toward the higher hiding potential. These latter notions of long and short are directly connected to processing time issues, as well to issues of the engineering resources needed to carefully quantify such parameters. Developing the working model of the y-axis will inevitably entail one part theory to one part picky-artist-empiricism. As the parts of the hodge-podge y-axis become better known, they can splinter off into their own independent axes if its worth it.

The z-axis is the "with or against the grain" axis which is the full axis - as opposed to the other two half-axes. The basic idea is that a given input bump has a pre-existing bias relative to whether one wishes to encode a '1' or a '0' at its location, which to some non-trivial extent is a function of the reading algorithms which will be employed. These biases magnitude is semi-correlated to the "hiding potential" of the y-axis, and....fortunately.... can be used advantageously as a variable in determining what magnitude of a tweak value is assigned to the bump in question. The concomitant basic idea is that when a bump is already your friend or even your friend in a big way, then why mess with it much, whereas when it is your enemy or a big time enemy, then you want to squash it like a four year old discovering how flat slugs can get underfoot. The really cool thing here is that, in general, the latter squashing operation tends more toward a local blurring operation as opposed to a local sharpening operation, and thus has somewhat less visibility per numeric tweak unit.

The above general description of the problem should suffice for many years. Clearly adding in chrominance issues will expand the definitions a bit, leading to a bit more signature bang for the visibility, and human visibility research which is applied to the problem of compression can equally be applied to this area but for diametrically opposed reasons. Fascinating possibilities truly. But alas, I am required to crank out some pot-shot first system which needs must neglect vast areas of the above general arenas. Here are its principles.

For speed's sake, local hiding potential will be calculated only based on a 3 by 3 neighborhood of pixels, the center one being signed and its eight neighbors. Beyond speed issues, there is also no data or coherent theory to support anything larger as well. The design issue boils down to canning the y-axis visibility thing, how to couple the luminance into this, and a little bit on the friend/enemy asymmetry thing. My guiding principles to start are simply to make a flat region zero, a classic pure maxima or minima region a "1.0" or the highest value, and to have "local lines", "smooth slopes", "saddle points" and whatnot fall out somewhere in between. In other words, let's pull out the darts and throw a few and see if any land on the board.

The following code has six basic parameters that will be used:

- 1) luminance
- 2) difference from local average
- 3) the asymmetry factor (with or against the grain)
- 4) minimum linear funkiness factor (our crude attempt at flat v. lines v. maxima)
- 5) bit plane bias factor

- 6) global gain (the user's single top level gain knob)

Even this list above can get complicated in their inter-relations and especially in our current lack of experimental data to support various specific formulas.

- 1) luminance is straightforward
- 2) difference from local average

our first generation stuff since it will directly eb involved in reading signatures (assuming we don't get fancy phase-only reading algorithms going).

- 3) the asymmetry factor is a single scalar applied to the "against the grain" side of the difference axis of number 2 directly above, as well and being modified by the minimum linear funkiness factor below. (Certainly it can eventually become a function of other variables if and when data and theory supports such).
- 4) The minimum linear funkiness factor is admittedly crude but it should be of some service even in a 3 by 3 neighborhood setting. The idea is that true 2D local minima and maxima will be highly perturbed along each of the four lines travelling through the center pixel of the 3 by 3 neighborhood, while a visual line or edge will tend to flatten out at least one of the four linear profiles. (The four linear profiles are each 3 pixels in length, i.e., the top left pixel - center - bottom right; the top center - center - bottom center; the top right - center - bottom left; the right center - center - left center;). Let's choose some metric of "funkiness" or "entropy" as applied to the minimum value for our ultimate parameter to be used as our "axis". Cheers to she or he who will take all of this to the next levels of refinement.
- 5) The bit plane bias factor is an interesting creature with two faces, the pre-emptive face and the post-emptive face. In the former you simply read the unsigned image and see where all the biases fall out for all the bit planes, then simply boost the "global gain" of the bit planes which are in total, going against your desired message, and leave the others alone or even slightly lower their gain. In the post-emptive modification, you churn out the whole signing process replete with the pre-emptive bit plane bias and the other 5 parameters listed here, and then you e.g. run the signed image through heavy JPG compression AND model the "gestalt distortion" of line stream printing and which bit planes are struggling or even in error, you read the image and find out the bit plane bias, and you run through the process again if you have good data driving the beefing process you should only need to perform this step once, or, you can easily Van-Citterize the process (arcane reference to reiterate the process with some damping factor applied to the tweaks).
- 6) Finally, there is the global gain. The goal is to make this single variable be the top level "intensity knob" that the slightly curious user can adjust if they want to. The very curious user can navigate down advanced menus to get their epirmental hands on the other five variables here, and who knows what others in the future.

whew, that's the most commenting I've ever done. I must be getting old or maybe I'm just realizing it would be nice to leave a signpost or two in this first dart throwing.

```

*//
lum_gain = luminance_gain
lum_gain = lumgain_lut( (int)*pbump );

// find current differential between bump value and local average
// this one can generally make use of inter-DN lut's:
// in this case, down to 0.25 of a DN
local_avg = *p1 + *p2 + *p3 + *p4;
diff = *pbump * f4 - local_avg;
detail_gain = detail_lut( (int)( fabs( (double)diff ) ) );

// now calculate tweak based first on message, include asymmetric gain
if( *(pxor++) ){
    if(diff<0.0)asym_gain = asymmetric_gain;
    else asym_gain = fi;
    *ptweak = fi; // slip this one in here
}
else {
    if(diff>0.0)asym_gain = asymmetric_gain;
    else asym_gain = fi;
    *ptweak = -fi;
}

// funky time: minimum linear funkiness factor
// line 1
bottomfunkt = fabs(double)( *pbump - *(p1-1)) + fabs(double)( *pbump - *(p3+1));
// line 2
dtomp = fabs(double)( *pbump - *p1) + fabs(double)( *pbump - *p3);
if(dtomp < bottomfunkt)bottomfunkt = dtomp;
// line 3
dtomp = fabs(double)( *pbump - *(p1+1)) + fabs(double)( *pbump - *(p3-1));
if(dtomp < bottomfunkt)bottomfunkt = dtomp;
// line 4
dtomp = fabs(double)( *pbump - *p2) + fabs(double)( *pbump - *p4);
if(dtomp < bottomfunkt)bottomfunkt = dtomp;
funky_gain = funky_lut( (int)bottomfunkt );

```

```

// add in the bias
// *ptweak += bit_bias*(pbit++);

// now put them all together somehow, but how??
gain = global_gain * (lum_gain + asym_gain * (funky_gain + detail_gain));
*ptweak *= gain;

// then add in subliminal grid
// eventually make this subject to local gain as well
if (gain > GRID_MINIMUM_GAIN) *ptweak += *psubliminal_grid;
psubliminal_grid += *ptweak++; *pbump++; *p1++; *p2++; *p3++; *p4++;

// load output array (tweak, sdata out (i*bump_size*Original_xdim*xdim,
// sdata (i*bump_size*Original_xdim*xdim, xdim, zdim, bump_size, jump_x);

// optionally JPEG compress (or whatever compress) the output buffer
// find the new bit biases, fine tune the bit bias values and
// repeat the above operations

delete () bit_bias;
delete () bump0;
delete () bump1;
delete () bump2;
return (1);
}

// sign_public_generation_1()
// =====
// int sign_public_generation_1()
// unsigned char *data, // input data to be signed
// long xdim, // it's x dimension
// long ydim, // it's y dimension
// long zdim, // generally 1 for RAW and 3 for 3x8bit RGB, data assumed R-G-B
// long bump_size, // number of pixels per singular bump along one dimension, e.g. 2 for 2x2
// bump_size // either 0 or 1, inefficient but simple
// unsigned char *message, // length of message in bits, also length of message string
// long message_length, // this is the separate "always gotta be there" message
// long control_message_length, // its length
// float *luminance_lut, // look up table mapping the scaling to luminance values
// float *detail_lut, // look up table mapping the scaling to local detail
// float *subliminal_grid, // this is the image of the subliminal grid, in the image domain
// unsigned char *data_out, // signed output data in same length and format as input, NULL if output
// is to be placed into input array 'data'
// float global_gain,
// float asymmetric_gain
// ) {
// long block_pixel_dimension, x_blocks, x_leftover, y_blocks, y_leftover, i, j, status=1;
// long temp_block_xdim, block_ydim;
// unsigned char *pdata, *pdata_out;

// block_pixel_dimension = SIGNATURE_BLOCK_DIMENSION * bump_size; // actual pixel dimension of a
// standard signature block
// x_blocks = 1+(xdim-1)/block_pixel_dimension; // number of full (and possibly partial on the last)
// basic blocks
// x_leftover = xdim%block_pixel_dimension - xdim*bump_size; // ignore fractional bumps on ends
// y_blocks = 1+(ydim-1)/block_pixel_dimension;
// y_leftover = ydim%block_pixel_dimension - ydim*bump_size; // ignore fractional bumps on ends
// though the straggly bits on the ends can cause a bit of a bookkeeping issue, they save alot of
// headaches when it comes time to write simple core algorithms sans if statements

// load the message length into the 16 bit long control message
int ii = 1;
control_message_length = 16;
for (i=0; i<16; i++) {
    if (i & (short)message_length) control_message[i] = 1;
    else control_message[i] = 0;
    if (*++i,
}

// BE SURE TO COPY END FRACTIONAL BUMP DATA FROM INPUT TO OUTPUT, UNCHANGED
// in other words, if xdimbump_size or ydimbump_size is non-zero, then we can
// immediately copy the leftmost and bottommost strip into the output buffer, unchanged
if (data_out != NULL) { // if data output buffer is the input buffer, no need for copying
    if (temp = (xdim*bump_size)) {
        for (i=0; i<ydim; i++) {
            pdata = sdata+(xdim*(i+1)*xdim-temp);
            pdata_out = sdata_out+(xdim*(i+1)*xdim-temp);
            for (j=0; j<temp*xdim; j++) *pdata_out++ = *pdata++;
        }
    }
    if (temp = (ydim*bump_size)) {
        pdata = sdata+(ydim-temp)*xdim*xdim;
        pdata_out = sdata_out+(ydim-temp)*xdim*xdim;
        for (i=0; i<temp*xdim*xdim; i++) *pdata_out++ = *pdata++;
    }
}

```